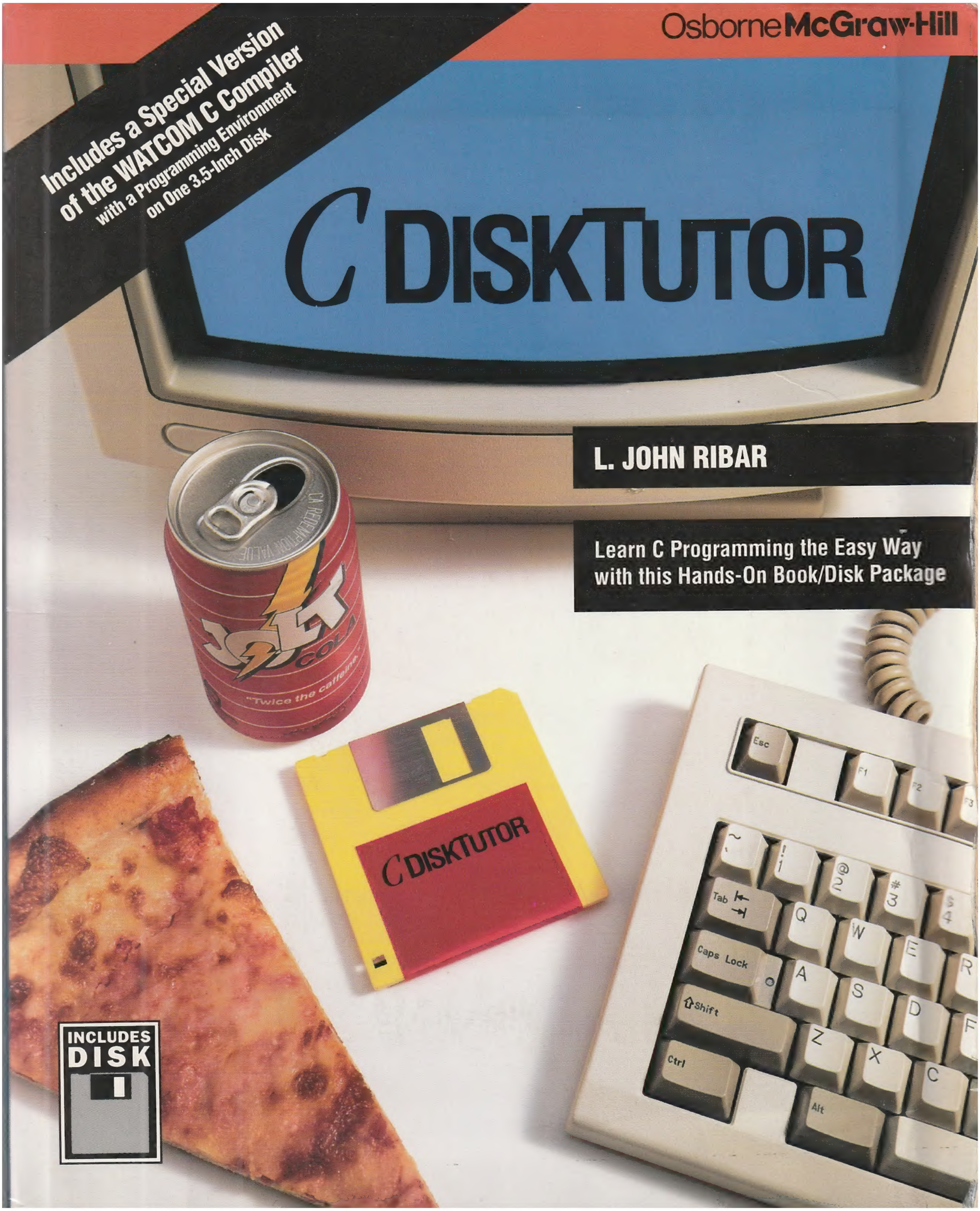
Osborne McGraw-Hill

Includes a Special Version of the WATCOM C Compiler with a Programming Environment on One 3.5-Inch Disk

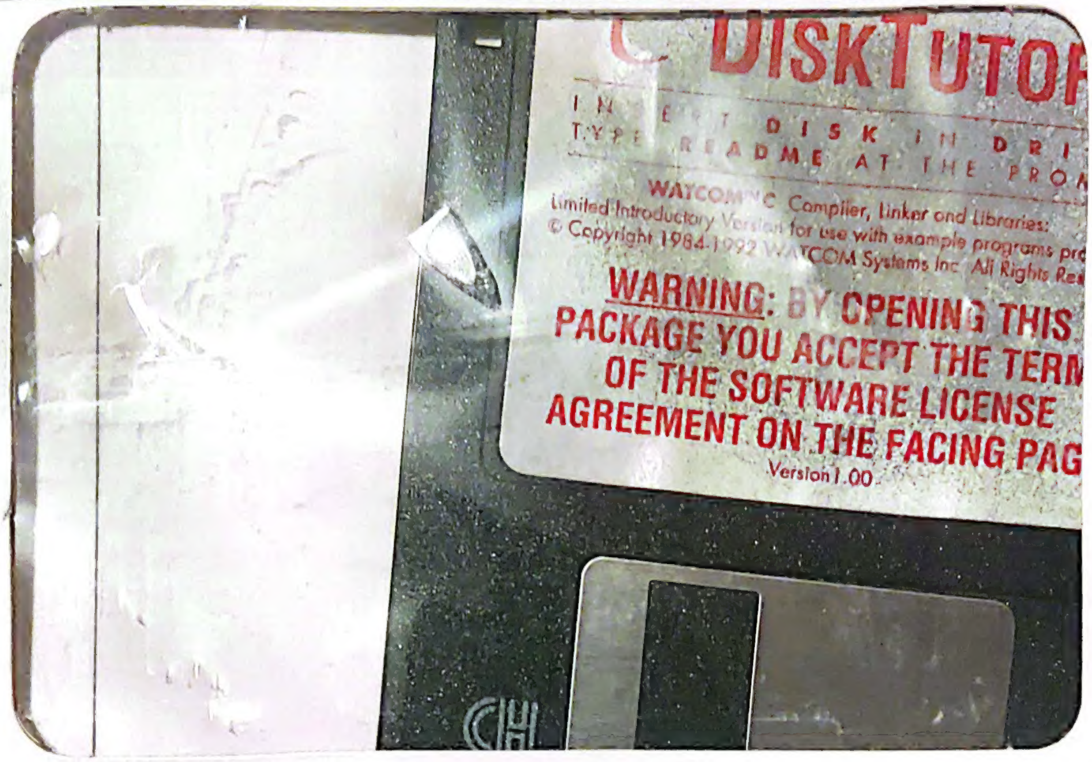
C DISKTUTOR

L. JOHN RIBAR

Learn C Programming the Easy Way with this Hands-On Book/Disk Package

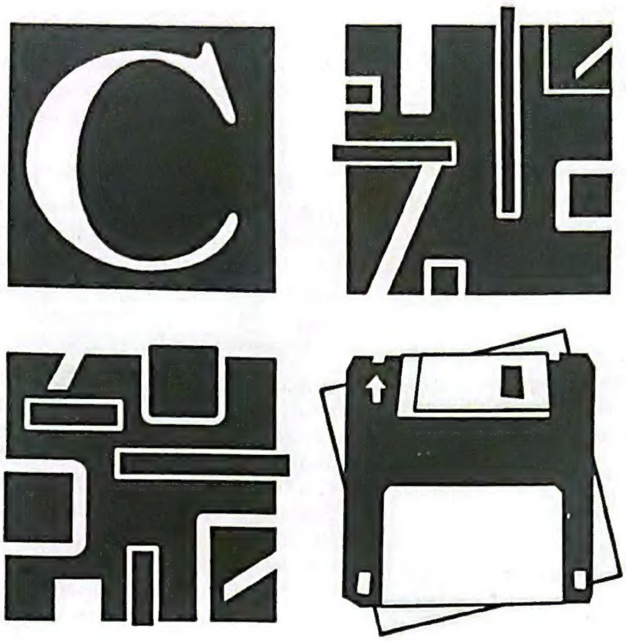


CUT TO OPEN



F
D
E

C DISKTUTOR



About the C DiskTutor Software

C DiskTutor Disk Contents: A special version of the highly acclaimed WATCOM C (including COMPILER, LINKER, and RUN-TIME LIBRARIES), an EDITING ENVIRONMENT, and all the programs in *C DiskTutor.* The software provided is all you will need to run every program in this book. The version of WATCOM C supplied limits program code and data each to 64 kilobytes of memory, including run-time libraries, and is intended to be introductory. (You can save money on an upgrade to the full WATCOM C package using the coupon at the back of this book.)

System Requirements: IBM PC or compatible Intel 8086 machines or higher. MS-DOS 3.0 or higher. 640K of memory and at least 2 1/2 MB disk space.

WARNING: BEFORE OPENING THIS PACKAGE, CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS OF THIS AGREEMENT WHICH INCLUDE THE SOFTWARE LICENSE (the "Agreement").

WATCOM Software License Agreement

BY OPENING THIS SEALED DISKETTE PACKAGE YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT YOU ARE NOT ENTITLED TO OPEN THIS DISKETTE PACKAGE AND YOU ARE NOT ENTITLED TO USE THE ENCLOSED SOFTWARE.

LICENSE

The enclosed software product (the "Software") is owned by WATCOM Systems Inc. ("WATCOM") or by its suppliers and is protected by copyright law and international treaty provisions. WATCOM hereby grants you a non-exclusive license to use the Software as set forth below:

You may:

> Use the Software only on one computer; make one copy of the Software for archival or backup purposes; permanently transfer your rights hereunder provided you also transfer all copies of the Software and provided the transferee agrees to the terms and conditions of this Agreement.

RESTRICTIONS

You may not:

> Use, copy, modify, or transfer the Software, or any copy in whole or in part, except as expressly provided for in this license; allow the Software to be used on more than one computer at a time or by more than one person at a time; rent or lease the Software; modify or adapt the Software in whole or in part including but not limited to translating or creating derivative works; disassemble or reverse compile for the purpose of reverse engineering the Software.

If the product package contains both 3-1/2" and 5-1/4" disks, you are only licensed to use one set. You may keep the other set for archival purposes but you may not otherwise use or allow others to use them except as expressly permitted in this Agreement.

SELECTION AND USE

You assume full responsibility for the selection of the Software to achieve your intended results and for the installation, use and results obtained from the Software. **WATCOM does not warrant that the functions contained in the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. Because of the inherent complexity of computer software you are advised to verify your work.**

TERMINATION

This Agreement and your license to use the Software will automatically terminate if you fail to comply with any provision of this Agreement. Upon termination you shall destroy all copies of the Software.

LIMITATIONS OF REMEDIES AND LIABILITY

WATCOM disclaims all warranties, oral or written, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose. No modification or addition to this warranty is authorized unless it is set forth in writing, references this Agreement and is signed on behalf of WATCOM by an authorized official.

In no event will WATCOM or its suppliers be liable for any other damages whatsoever including, but not limited to, direct, indirect, special, incidental, or consequential damages or other pecuniary loss arising out of the use of or inability to use the Software, even if WATCOM has been advised of the possibility of such damages. In particular, WATCOM and its suppliers are not responsible for any costs including, without limitation, loss of business profits, business interruption, loss of business information, the cost of recovering such information, the cost of substitute software, or claims by third parties. **In no case shall WATCOM's liability exceed the amount of the license fee actually paid by you.**

This Agreement gives you certain legal rights. You may have others under law, so some of the above may not apply.

GENERAL

This Agreement shall be governed by the laws of the Province of Ontario and the laws of Canada therein.

DESCRIPTION OF THE SOFTWARE

The Software is a limited, introductory version of WATCOM C for IBM PC compatibles with DOS. A compiler, linker and run-time libraries are included. The Software is intended to enable the user to run the example programs provided but is limited in capacity (program size) and also restricted to the small memory model. Users may wish to upgrade to the full WATCOM C package which is detailed elsewhere in this book.

C DISKTUTOR

L. John Ribar

Osborne McGraw-Hill

Berkeley New York St. Louis San Francisco
Auckland Bogotá Hamburg London Madrid
Mexico City Milan Montreal New Delhi Panama City
Paris São Paulo Singapore Sydney
Tokyo Toronto

Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations or book distributors outside of the U.S.A., please write to Osborne McGraw-Hill at the above address.

C DiskTutor

1234567890 DOC 998765432

ISBN 0-07-881798-6

To Mom and Dad, for helping me learn to be creative, to reach for the stars, and to stand behind whatever I want to do.

Publisher
Kenna S. Wood

Editor-In-Chief
Jeffrey M. Pepper

Acquisitions Editor
William Pollock

Associate Editor
Vicki Van Ausdall

Technical Editor
Greg Walters

Project Editor
Erica Spaberg

Copy Editor
Carol Henry

Proofreaders
Audrey Johnson
Colleen Paretty

Editorial Assistant
Judith A. Kleppe

Cover Designer
Graphic Eye, Inc.

Illustrator
Susie C. Kim

Director, Manufacturing & Production
Deborah Wilson

Production Supervisor
Barry Michael Bergin

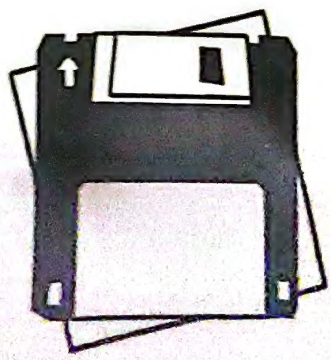
Quality Control Specialist
Bob Myren

Computer Designer
Stefany Otis

Typesetters
Jani Beckwith
Helena Charm
J. E. Christgau
Marcela Hancik
Peter Hancik
Susie C. Kim
Fred Lass
Lance Ravella
Michelle Salinaro
Marla Shelasky

Manufacturing Assistant
George Anderson

Production Coordinator
Linda Beatty



C O N T E N T S

C O N T E N T S

C O N T E N T S

ACKNOWLEDGMENTS

A lot of people deserve a great deal of thanks for their help and support in completing this book under a very hectic schedule (and during the holidays, no less):

Greg Walters, for his friendship, detailed technical review, Pascal example code, and help with the ANSI and ASCII tables, which helped me make this book as strong as it is.

Bill Pollock, Vicki Van Ausdall, and Erica Spaberg, for editing this so many times that I started to understand what they were doing (and they even started learning C!).

Jeff Pepper for getting me into this in the first place, and Phillippe Kahn for keeping my conviction strong.

My C class (Carol Park, Charlie Connahan, Rick Graham, Kim Speicher, Bill Collier, and the rest) for helping me find the things that I forgot to teach.

The family of SOCO, for their support through all of this.

Of course, the biggest thanks to Deborah, Louis, Jamie, Michael, and Leah, for their love and support. I know I was really busy during Christmas, but look what we did! Thanks.

INTRODUCTION

Welcome to the C language! This special book and disk package has been designed to help you learn the C programming language quickly and simply, without the need for other programs, compilers, editors, or disks. In addition, the programs and functions presented in this book will be useful in your C programming projects for years to come.

Included with this book is a disk containing a complete C compiler (created by WATCOM); all the source code from this book; and a special DiskTutor programming environment (DTE) that allows you to edit, compile, and run your C programs interactively. Altogether, and when used in conjunction with this book, these tools are designed to help you learn standard C programming.

Once you have mastered this book, if you want to continue developing C programs, consider purchasing a professional-quality compiler and development environment to support your work. For instance, the WATCOM compiler and DTE included with this book both have professional versions available; coupons for ordering them are at the back of the book.

HOW THIS BOOK IS ORGANIZED

This book is divided into four parts: The first part helps you install and learn how to use the C DiskTutor compiler and DTE.

The second part of the book explains the basics of C language programming. As you complete each chapter, you'll find Mini-Case and Case Studies—example programming assignments you should complete to aid your understanding of the concepts you are learning.

The third part of the book shows completed C programming examples for your study. These are the final programs developed from the Mini-Case and Case Studies introduced in Part Two.

The fourth part of the book contains reference materials for the compiler.

CONVENTIONS USED IN THE BOOK

In general, the following typeface conventions are used throughout the book:

Boldface	C function names
Italic	Used to introduce new concepts, or generic terms used in portions of code

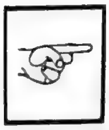
Monospaced text is used to denote pieces of C code, as shown below:

```
#include <stdio.h>

main()
{
   printf("This is a test\n");
}
```

Many special paragraphs are indented and marked with icons to capture your attention. The following types of markings are used:

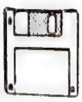
Note: Note paragraphs tell you to take notice of something that is going on.



Tip: Tip paragraphs let you know about special features of the C language that are used by many practicing C programmers. These tips can help you save time as you write your own C programs.

Caution: The caution paragraphs discuss potential problems that can occur in your programming. These are especially important to read as you progress through the book.

In addition, when you see a disk icon next to a program you know there is a file on the *C DiskTutor* disk corresponding to the program shown in the book. The disk icon is shown here:



HOW TO LEARN C

As you work through this book, make use of the DiskTutor Environment (DTE) and compiler to try out the code examples given in the book and on the disk. The more work you do with C, the more you will pick up the language. Feel free to experiment at any time! The best way to learn C programming is by trying out your own ideas, once you have learned the basics.

When you feel confident with the C concepts presented here, use the included Screen Library and Case Studies to create professional-quality programs in minimal time. The concepts and code you develop using this book will prove useful over and over in the development of your future C programs!

PART ONE

GETTING STARTED WITH *C DISKTUTOR*

C H A P T E R

INSTALLATION AND SETUP

C *DiskTutor* is a special book-and-disk package that will quickly bring you up to speed as a C programmer. As part of this package, the enclosed disk contains

- A complete C language compiler
- All the programs discussed in this book
- The C DiskTutor Environment (DTE)—an editor that helps you edit, compile, and run programs

The best way to learn C is by writing C programs. Therefore, before you start your quest to learn C, you'll need to install the C DiskTutor from the disk that comes with this book. Here in Chapter 1 you will read step-by-step instructions on how to install the disk, as well as how to use the DiskTutor Environment.

INSTALLING THE C DISKTUTOR

The C DiskTutor software requires approximately 2.5 megabytes of free disk space. It cannot be run from a floppy disk drive.

To simplify the installation of the DiskTutor, a special installation program has been included on the disk. It installs all of the necessary files on your hard disk, and then modifies your system's AUTOEXEC.BAT and CONFIG.SYS files as needed to properly run the DiskTutor.

USING THE INSTALL PROGRAM

To begin the installation procedure, put the C DiskTutor disk into your floppy drive. In the instructions that follow, it is assumed that the DiskTutor disk is in drive A.

The first step in the installation is to start the installation program, INSTALL.EXE—we'll call it simply Install. From the DOS prompt on drive C, type **A:INSTALL** (or, if the C DiskTutor disk has been inserted in drive B, type **B:INSTALL**).

You will be greeted by a screen like the one in Figure 1-1. After reading the information on the screen, press any key (except ESC) to continue. If you want to cancel the installation, press ESC.

Next, specify the hard disk on which you want the DiskTutor software and files to be installed. The disk selection screen (Figure 1-2) will show only the drives that are available on your system. Use the arrow keys to select the drive you want, and then press ENTER.

Finally, select the base directory that will store your C DiskTutor files. As you can see in Figure 1-3, the Install program suggests you use \CDT as the base directory. If you want to use another directory, type the new directory name in at this time, and press ENTER.

The Install program will create the following directory tree on the drive and directory you have specified (this example assumes you chose \CDT as the base directory).

```
\CDT
    \CHAP2
    \CHAP3
    \CHAP4
    \CHAP5
    \CHAP6
    \CHAP7
    \CHAP8
    \CHAP9
    \CHAP10
    \CHAP11
    \CHAP12
    \CHAP13
    \CHAP14
    \CHAP15
    \BIN
    \LIB286
    \H
```

```
Welcome to the C DiskTutor disk, supplied with the C DiskTutor
book published by Osborne/McGraw-Hill.  Copyright 1992.

This program will install C DiskTutor version 1.0
on your computer system and verify the integrity of the
distribution disk(s).  You may press the [Esc] key at any time to
abort the installation.

INSTALL will ask you several questions about your computer
hardware. Each question has a default answer.  If the default answer
is correct, press the ENTER key in response to the question.
Otherwise, type the answer and then press the ENTER key.

If you make a mistake while typing, press the BACKSPACE key
and then retype the answer.

Press [Esc] to quit, any other key to continue ...
```

FIGURE 1-1

The opening installation screen

```
On which disk drive do you wish to install C DiskTutor?

                 Drive A:
                 Drive B:
                 Drive C:
                 Drive D:
                 Drive F:
                 Drive X:
                 Drive Y:
                 Drive Z:
```

FIGURE 1-2

The disk selection screen

```
Now you need to specify the disk destination subdirectory.  The
subdirectory is the location on your disk where the C DiskTutor system
will be installed.  If you don't know what a "subdirectory" is, you
probably need not worry about it, and should press the ENTER or
RETURN key for the next   prompt.

Later in the installation, you can optionally allow Install to
modify your AUTOEXEC.BAT file to give you access to the
subdirectory selected.

Which subdirectory ([Enter] = \CDT) ?
\CDT
```

FIGURE 1-3

Selecting a directory for installation

Within this base directory structure are the programs that you will be studying in each chapter of this book. For example, you can find the programs for Chapter 2 in the \CDT\CHAP2 subdirectory. The subdirectories named BIN, LIB286, and H contain files used by the C DiskTutor compiler.

At this point, Install begins to install all the necessary files to the drive and directory you have specified. You will see a great number of files scrolling by as this happens (see Figure 1-4).

Finally, Install asks you if the CONFIG.SYS and AUTOEXEC.BAT files should be modified automatically, as explained in the next section.

MODIFYING CONFIG.SYS AND AUTOEXEC.BAT

Once all the C DiskTutor files have been installed on your hard disk drive, your system's AUTOEXEC.BAT and CONFIG.SYS files may need to be updated. In the CONFIG.SYS file, the FILES statement parameter must be at least 20, like this:

```
FILES=20
```

```
      Reading: Library File --> 41
    Verifying: 41
      Reading: Library File --> 51
    Verifying: 51
      Reading: Library File --> 61
    Verifying: 61
      Reading: Library File --> 71
    Verifying: 71
      Reading: Library File --> 81
    Verifying: 81
      Writing: C:\CDT\CHAP5\P5-1.C
       Making: CHAP5
Decompressing: C:\CDT\CHAP5\P5-1.C
      Writing: C:\CDT\CHAP5\P5-2.C
Decompressing: C:\CDT\CHAP5\P5-2.C
      Writing: C:\CDT\CHAP5\P5-3.C
Decompressing: C:\CDT\CHAP5\P5-3.C
      Writing: C:\CDT\CHAP5\P5-4.C
Decompressing: C:\CDT\CHAP5\P5-4.C
      Writing: C:\CDT\CHAP5\P5-5.C
Decompressing: C:\CDT\CHAP5\P5-5.C
      Writing: C:\CDT\CHAP5\P5-6.C
```

FIGURE 1-4
Files being installed by the Install program

The Install program's automatic update of CONFIG.SYS is based on the following decisions:

- If there is no CONFIG.SYS file, a new file is created.
- If CONFIG.SYS exists but does not contain a FILES statement, the appropriate statement is inserted in the file.
- If CONFIG.SYS does contain a FILES statement, it is modified (if necessary) to FILES=20.

Install will automatically modify the AUTOEXEC.BAT file, too, as necessary. First, the \CDT\BIN directory is added to the PATH statement in AUTOEXEC.BAT (of course, this directory name will reflect whatever base directory you specified previously). Then the following two lines are inserted, if they don't already exist in the AUTOEXEC file:

```
SET INCLUDE=C:\CDT\H
SET WATCOM=C:\CDT\
```

Again, as with the CONFIG.SYS file, Install will work correctly whether or not the AUTOEXEC.BAT file exists, and whether or not it contains the correct PATH, SET INCLUDE, and SET WATCOM statements.

The C DiskTutor installation is now complete. At this point you must reboot your computer so that the changes made to AUTOEXEC.BAT and CONFIG.SYS (if any) can take effect.

USING THE DISKTUTOR ENVIRONMENT

C DiskTutor comes with a program called DiskTutor Environment (DTE.EXE), which is a programmer's editor specifically designed to work with the compiler and source files provided on the C DiskTutor disk. As you work through this book, you will edit program files, compile them, and run the resulting programs—all without ever leaving the Environment.

This section explains how to use DTE. Before beginning, you should have already installed the C DiskTutor disk, as discussed in this chapter.

STARTING DTE AND OPENING A FILE

To start the DiskTutor Environment, type **DTE** at the DOS command line. You will be greeted by the opening screen, illustrated in Figure 1-5. Press ENTER to begin the program.

> **Tip:** The instructions in this chapter tell you how to work within DTE using your keyboard, but you can also use your mouse. With a mouse, simply point to an option on the screen, and click the left button to select it. Generally, the mouse and keyboard can be used interchangeably to select commands.

Now that you have the Environment open on your desktop, let's see how to use the menu bar at the top of the screen, and the menus it presents for your selection. Press ALT-F to pull down the Files menu. There you'll see the options for working with files. To create a new file, you'll choose New; to open an existing file, you'll choose Open. When you choose the Open option, you will see the file selection dialog box shown in Figure 1-6.

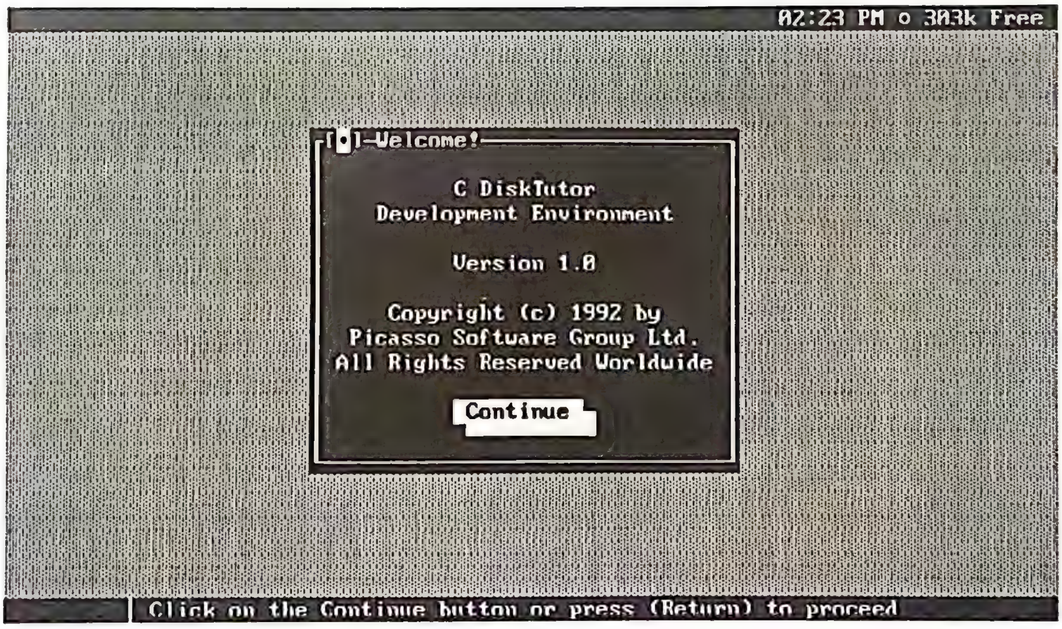


FIGURE 1-5
The opening DiskTutor Environment screen

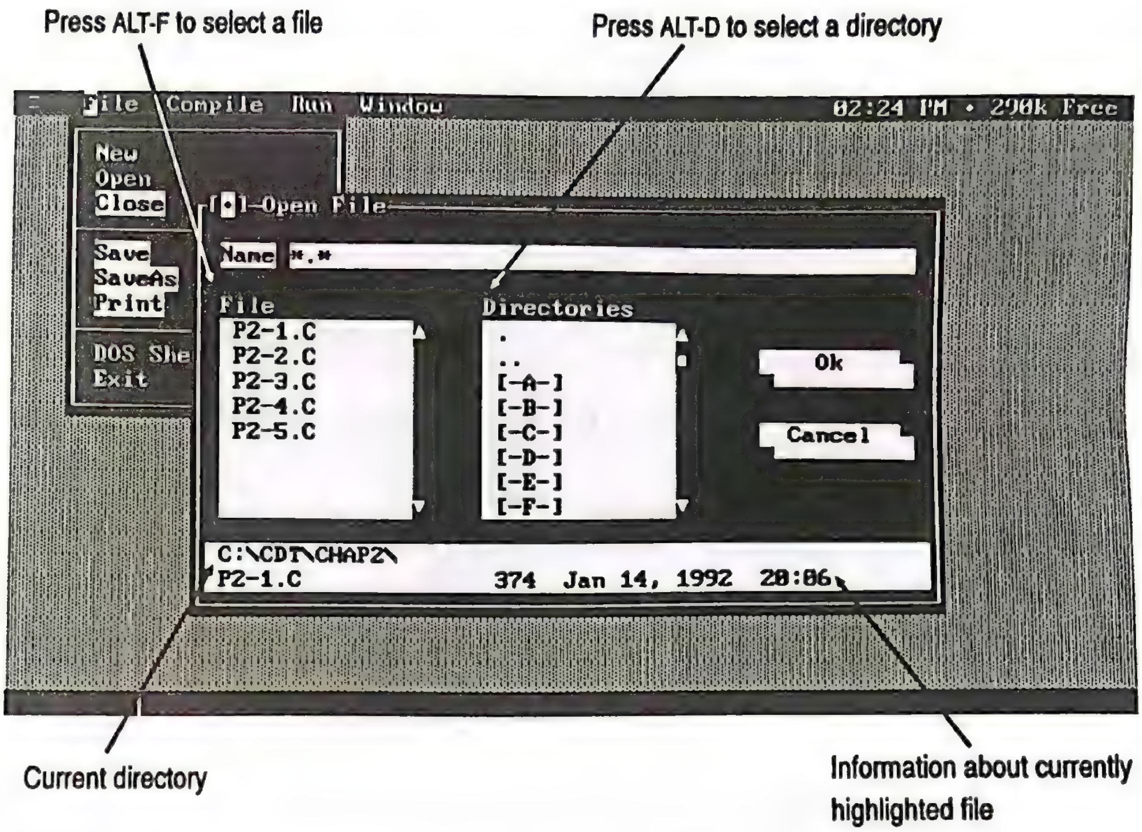


FIGURE 1-6
The file selection dialog box

Inside this dialog box (and any other dialog boxes in DTE), you will see several letters that are different colors (or a different shade on monochrome monitors). For instance, in Figure 1-6 notice at the very top of the list of filenames that the *F* in *File* is a different shade. This tells you that pressing ALT plus the F key (ALT-F) will execute that function. Thus ALT-F selects a file, ALT-D selects a directory, ALT-N lets you type a filename in the Name field, and ALT-C cancels the File Open function. When you've made your selection, press the ENTER key to execute it.

With a mouse, you select a directory and/or file by pointing to your selection and clicking the left mouse button, and then clicking on the OK button.

Once you have selected a directory and/or file, you will see a window containing the file selected, as shown in Figure 1-7. (If you chose File New, the file window will be empty.)

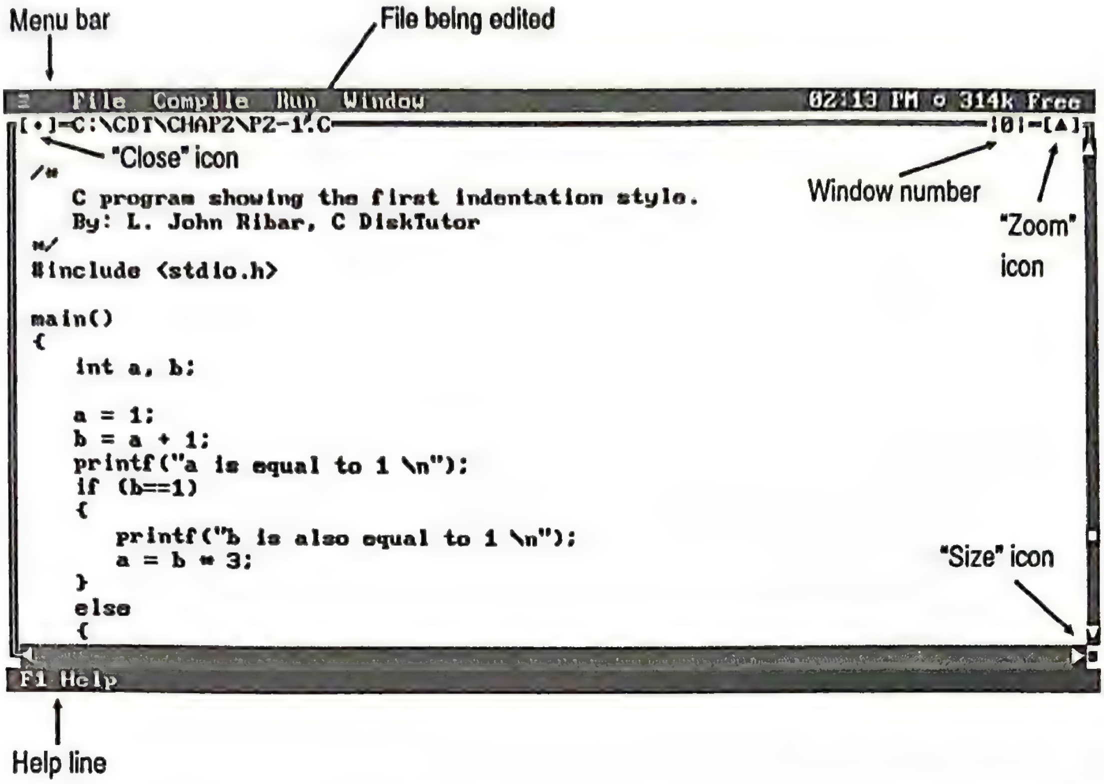


FIGURE 1-7
A file displayed in a window

You may open up to ten windows, with different files, by using the File Open or File New menu options. Each window has a number in the upper-right corner (0 through 9). To quickly move to a specific window, hold down the ALT key and press the number of the desired window.

EDITING FILES

Editing files is as simple as typing the information you want to be in the file. When you have completed your editing, pull down the File menu again (ALT-F), and choose Save (or SaveAs for new files) to store your file on disk.

The DTE editor allows the entry of text in a manner similar to most word processing programs available on computers. As you type, your text

will be displayed in a window. At the end of each line, the text will wrap to the next line, if the line of code is longer than the width of the window.

The BACKSPACE and DEL keys are used to delete characters. You can use the arrow keys, PGUP, PGDN, HOME, and END to move around within your file.

Pressing F1 at any time will display the context-sensitive help, telling you the options available at that particular point in the program.

The CR (carriage return) key is used to begin new paragraphs, or to break a long line into two shorter lines.

The TAB key moves across the screen in 3-character increments, and is used to align blocks of code and comments.

COMPILING AND RUNNING PROGRAMS

Compiling programs is simple—just press ALT-C to execute the Compile option in the menu bar. This compiles the program file you are editing in the current window. Then press ALT-R to run it.

DISKTUTOR ENVIRONMENT MENUS

The following sections describe each option available from within DTE's menus.

THE SYSTEM MENU (≡)

You can access the System menu by clicking on the (≡) symbol in the menu bar, or by pressing ALT-SPACEBAR. The System menu has two options: Setup, which allows you to configure DTE (its colors, menu types, and so on); and About, which tells you about the current version of the DTE program (copyright, version number, and so on).

THE FILE MENU

The File menu has several options. Some of these are always available (New, Open, DOS Shell, and Exit), and others are only available if one or more files are already open. When options are not available for selection, they are displayed on the menu in a lighter color ("ghosted").

New Opens a new file for editing.

Open Opens an existing file for editing, compiling, running, or viewing.

Close Closes the current file and the window that holds it.

Save Saves the current file.

SaveAs Saves the current file and lets you give it a new name.

DOS Shell Exits temporarily to the DOS shell. You can type **EXIT** to return to the DTE.

> **Warning:** When you exit to the the DOS Shell, do not run any programs with TSR (terminate and stay resident) capabilities. These programs, including the DOS PRINT command, must be run *before* starting DTE for the first time. Otherwise, your system may crash when it returns to DTE.

Exit Exits the DTE program.

THE COMPILE MENU

The Compile menu has two options: Compile and Build.

Compile Lets you compile the current program file.

Build Compiles the current file, and then links the file for running at the DOS prompt. Use Build on a program file before running it using the Run menu. Build also lets you specify the names of additional files that are to be included in the compilation. For instance, when you start using the Screen Library in Chapter 8, you will need to specify the Screen.C file using the Build option, so that the screen library functions will be available to your programs.

THE RUN MENU

You use the Run menu to begin execution of a program after it has been compiled and/or built using the Compile menu. The Run menu runs the program at the DOS command line, and waits for you to press a key before returning to the DTE.

THE WINDOW MENU

The Window menu has several options that modify the window layout on your screen. These options are most useful when you have two or more windows open at one time.

> **Tip:** To quickly move between windows, you can press ALT plus a number key (0 through 9) to switch to any of the first ten files opened.

Zoom This option fills the entire screen with the current window, allowing you to see more of the file within that window. You can also zoom with the mouse; point to the Zoom icon (the triangle symbol) in the upper-right corner of the window, and click the left mouse button.

The Zoom option is a toggle, meaning that each time you select it on the menu, the window will either go to full-screen size or back to its original size. To do this with the mouse, click on the triangle in the upper-right corner of the window.

Move Use this to move the current window using the arrow keys on the keyboard. To do this with the mouse, click on the top border of the window (near the filename) with the left mouse button, and hold the button down as you move the window to the desired location on the desktop.

Size This option lets you resize a window using the arrow keys on the keyboard. To do this with the mouse, click on the small square at the lower-right corner of the window, and then move the mouse until the window is the size you want it to be.

Close This closes the window; it performs the same function as the Close option on the File menu. To close the window using the mouse, click on the [.] icon at the top-left corner of the window.

List Windows Displays a list of currently open windows, and allows you to quickly move to another window. You can select a window using the arrow keys, and then pressing ENTER. With a mouse, simply click on the desired window.

IndexCard, Pyramid, and Waterfall Choose one of these window arrangement options to reorganize the windows open on your desktop. If none of

these arrangements is to your liking, you can arrange the windows yourself using the Move and Size options.

LEARN BY DOING

As mentioned at the start of this chapter, the best way to learn C programming is by doing it. As you go through the book, you will examine many examples of valid C programs. But for real practice, most chapters also include mini-cases or case studies.

You'll find the *mini-case studies* in the earlier chapters; these are assignments you can use to strengthen your C programming skills as you learn them. The mini-cases give you small programs to write, to test what you have learned and to allow you some flexibility in your progress. Sample solutions to the mini-cases are given in Chapter 12, in case you need a jump-start.

Case studies are larger programming projects that give you a chance to really test your learning, and to use your imagination. There are three case studies, presented in Chapters 8, 10, and 11 of the book, and their solutions are given in Chapters 13, 14, and 15.

PROGRAM FILES WITH MULTIPLE NAMES

Some of the program files created in the later chapters of this book have two names. Starting in Chapter 1, each file on the C DiskTutor disk has a name in this format:

```
P<chapter>-<file number>.C
```

where P stands for Program, *<chapter>* is the number of the chapter where the code first appeared, and *<file number>* is the sequential number of the program within each chapter. Thus the first program in Chapter 2 is P2-1, and the third file in Chapter 5 is P5-3.

In later chapters, on the other hand, program files are developed that can be used in many places. For instance, a screen-handling library is

presented in Chapter 8. For these programs, you'll find a batch file called NAMED.BAT in the directory of files for that chapter. When executed, this batch file copies the files with numbered filenames into files that have more descriptive names. For example, P8-1.C might get copied to a file named Screen.C, so you'll have a more descriptive name of what the file contains.

To determine if there is a NAMED.BAT batch file in a particular chapter's directory, use the DIR command at the DOS command line, and look for a file named NAMED.BAT. If it's there, type NAMED at the command line; this executes the batch file and creates the files you will use in your programming tasks. This is best done before you start working in a specific chapter.

The numbered filenames let you quickly identify the chapter in which a particular program is discussed, and the more descriptive filenames let you use the programs later, in real applications. The comments at the top of these program files show both filenames, for a handy cross-reference.

SUMMARY

- The Install (INSTALL.EXE) program on the C DiskTutor disk performs all the work needed to install the compiler, the C DiskTutor Environment (DTE), and the program source files (files containing C source code for each program).
- The DTE program allows you to write, compile, and run programs from within one integrated programming environment.
- Mini-case and case studies contain programming projects for you to complete, so that you can test your progress and use your imagination.
- Some program files have two names—one name helps you locate the file in a particular chapter of this book; the other name is more descriptive, and tells you the purpose of the program.
- In Chapter 3, you will get a complete, step-by-step trial in using the DTE program and writing your first C program.

C H A P T E R

PROGRAM DESIGN

Before beginning your C language tutorial, you need some background information about the process of developing a program. An in-depth study of this process is beyond the scope of this book, of course, but as you start programming in C, it will be useful to know some basic design principles. If you are already a practicing programmer, or have had software engineering training, you may only need to skim this chapter before proceeding.

There are two important topics you'll need to explore before beginning this tutorial: programming style, and the steps involved in the development process itself.

PROGRAMMING STYLE

Programming style is the manner in which the code is constructed or formatted in the file. Programming style is extremely important when you are working with the C language. Because C is so flexible when it comes to the format of your code, requiring no specific layout, you need to train yourself to follow some guidelines. Each

C H A P T E R

PROGRAM DESIGN

Before beginning your C language tutorial, you need some background information about the process of developing a program. An in-depth study of this process is beyond the scope of this book, of course, but as you start programming in C, it will be useful to know some basic design principles. If you are already a practicing programmer, or have had software engineering training, you may only need to skim this chapter before proceeding.

There are two important topics you'll need to explore before beginning this tutorial: programming style, and the steps involved in the development process itself.

PROGRAMMING STYLE

Programming style is the manner in which the code is constructed or formatted in the file. Programming style is extremely important when you are working with the C language. Because C is so flexible when it comes to the format of your code, requiring no specific layout, you need to train yourself to follow some guidelines. Each

programmer develops his or her own style of programming as he or she grows familiar with any programming language. This is especially important in a language with as much flexibility as C has—if you don't set up some rules for yourself, you may have trouble reading your own code.

The most important part of good programming style is just being consistent in the ways you indent lines, format comments, and assign variable names. The following sections offer examples of various programming styles. Use a style that suits you, or modify one to make your own—but be consistent! Consistency in your programming style will prove invaluable when you return later to reread your code, and will limit possible confusion for other programmers who examine your code.

> **Tip:** By the way, don't feel that you need to make decisions right now about your programming style. Try the different techniques shown in this chapter, and add changes as you see fit. As you learn to program, your style will continually evolve. As your style develops, however, do take care to ensure that it remains consistent within any one program or project.

INDENTATION

The C language does not require line formatting or indentation of any kind. Therefore, it's entirely up to you if, when, and how you indent. The example programs in this section show you how others have decided to format program lines with indentation.

Indentation shows how *blocks* of code are combined. In C, there are numerous ways to use blocks, and most blocks are surrounded by *braces*: { and }. As you study the examples that follow, remember to notice how the blocks are indented.

The number of spaces you choose to indent is also a matter of personal style; most programmers find that indenting two to four spaces per level of indentation is sufficient. Find a level of indentation that looks right to you, and use it consistently.

When to indent? One option is to indent each block of statements from the controlling part of the block. The braces, however, are not indented. This style is demonstrated as follows in Program P2-1.

P2-1

```
      /*
         C program showing the first indentation style.
         By: L. John Ribar, C DiskTutor
      */
      #include <stdio.h>

      main()
      {                  <----  Braces are
                                not indented
Indented --> int a, b;

         a = 1;
         b = a + 1;
         printf("a is equal to 1 \n");
         if (b==1)
Indented    {
within the -->  printf("b is also equal to 1 \n");
block           a = b * 3;
            }
Matching    else
braces are  {
indented the    printf("b is not equal to 1 \n");
same amount     b = a * 3;
            }
      }
```

A second indentation option involves indenting the braces along with the indented block of code. Program P2-2 is an example of this format:

P2-2

```
      /*
         C program showing the second indentation style.
         By: L. John Ribar, C DiskTutor
      */

      #include <stdio.h>

Braces are  main ()
indented -->   {
               int a, b;
Code is also
indented -->   a = 1;
               b = a + 1;
               printf("a is equal to 1 \n");
               if (b==1)
                  {
                  printf("b is also equal to 1 \n");
```

```
      a = b * 3;
      }
   else
      {
      printf("b is not equal to 1 \n");
      b = a * 3;
      }
}
```

Matching braces are indented the same amount

In a third indentation option, the first brace is moved up into the line of code that precedes the indented block. Program P2-3 is an example of this style.

P2-3

```
/*
   C program showing the third indentation style.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main() {
   int a, b;

   a = 1;
   b = a + 1;
   printf("a is equal to 1 \n");
   if (b==1) {
      printf("b is also equal to 1 \n");
      a = b * 3;
      }
   else {
      printf("b is not equal to 1 \n");
      b = a * 3;
      }
   }
```

Opening brace follows the opening of the block

Code is indented

Closing brace lines up with the code

Matching braces are *not* indented the same amount

All three of these indentation styles are perfectly acceptable—as long as they are used consistently within the program.

Note: The program examples throughout this book and on the disk included with this book all use the style shown in Program P2-1.

COMMENTS

Comments are very important in all programming, and especially with the C language. Unfortunately, they are often ignored or played down by programmers; don't make the mistake of thinking that comments are unimportant just because they are not part of the code necessary to make the program work.

> **Note:** Although comments are not required in C programs, try to get into the habit of commenting your code. Comments may not appear significant when you are writing a program, but they are very important when you return later to decipher what you have written.

Comments need not be lengthy, but they do need to explain what is going on, so that you, the programmer, or other users can understand the program in the future. It is not necessary, or even desirable, to comment on each line of code. You should, however, place comments in at least the following places:

- *At the beginning of the file* to describe what is in the file, perhaps its author and the date written, and any dependencies the program has on other files.
- *At the beginning of each function* to describe how to use the function, what it does, and what parameters it uses. (*Parameters* are the values or variables sent into and returned from functions.)
- *In areas of code where the algorithms are not straightforward,* to later help you remember why things were done in a particular way.

Comments start with two characters—a backslash and an asterisk, with no spaces between them, like this: /*. Comments end with the same asterisk and backslash characters, but in reversed order: */. Comments can be written on a single line, or can cross multiple lines. You can use any characters except additional comments; comments within comments are called *nested comments,* and are not allowed. The following example shows comments on a single line, and some that cross several lines:

```
/*
   Function: Check4Spaces()

   This function checks for space characters within a
```

```
    character array.

    Parameters:
    IN              char *txt   Text to check for spaces
    OUT             none

    By: L. John Ribar, C DiskTutor
*/
                Function name          Parameter

void Check4Spaces( char *txt )
{                          Parameter type
Declaration of   int i;                     /* counter variable */
a variable
    for (i=0; txt[i]!=0; i++)  /* for each character in
                                          the string */
    {
        if (txt[i] == ' ')      /* check for a space */
            printf("space found \n");
    }
}                               /* end of check4spaces */
```

The function in the foregoing listing prints a message each time a space is found in a character string. The code may not yet make sense to you, but notice how the comments help you to better understand what is happening. Also notice that the format of a comment can change from line to line, and how a comment can occupy one or more lines. You can format your comments as you choose, based on how you want your final program to look.

You may want to formalize the appearance of your comments, adding lines and positioning the comments to make their display more attractive, as shown here:

```
/*************************************************************
 *   Function: Check4Spaces()                                *
 *                                                           *
 *   This function checks for space characters within a      *
 *   character array.                                        *
 *                                                           *
 *   Written by:   John Ribar                                *
 *   Date:         December 12, 1992                         *
 *   System:       C DiskTutor                               *
 *                                                           *
 *   Parameters:                                             *
 *   IN            char *txt   Text to check for spaces      *
```

```
 *   OUT              none                                         *
 ****************************************************************/

void Check4Spaces( char *txt )
{
   int i;                                   /* counter variable */

   for (i=0; txt[i]!=0; i++)     /* for each character in */
   {                                           /* the string */
       if (txt[i] == ' ')            /* check for a space */
           printf("space found \n");
   }
} /********************************** end of check4spaces */
```

Tip: Comments that are boxed and aligned are very easy to read when it comes time to maintain (make changes to) your code. Just remember this extra "artwork" takes a lot of time, and is best saved for the final version of your program, when you've already taken care of other, more important changes.

VARIABLE NAMES

Just as there are many styles for indentation and comments, there are numerous variable-naming conventions. *Variables* are data storage locations within C programs. You should give variables names that will be simple to recall as you are programming. Several ideas for naming variables are shown in the table below. As with indentation and comments, the choice of a variable-naming style is up to you, as long as you use it consistently.

Give your variables names that are long enough to remind you of what the variables are used for, but not so long that you dread typing them when you need them in a statement. For instance, **interest**, **payment**, and **total** are better names for variables than **i**, **p**, and **t**. On the other hand, names such as **interest_rate_for_the_loan**, **payment_for_the_loan**, and **total_amount_paid_on_the_loan** make for a good deal of typing, and are unnecessary for most programs.

There are always exceptions—situations where you will in fact need a very long or very short variable name. Most compilers limit the length of variables to some extent; most limit you to 32 significant characters. The DiskTutor compiler allows variables of any length, but any characters after

the first 40 are ignored. Use your common sense, and C will allow you almost all the freedom you need.

Note: *Significant characters* are the characters at the beginning of a variable name that the compiler actually uses. For instance, the DiskTutor compiler allows 40 significant characters. This does not prevent you from using a 60-character name, but the compiler will just ignore the last 20 characters.

There are several capitalization conventions in C programming, as in other languages, as demonstrated in the examples that follow. Capitalization is important, because C is a *case-sensitive* language. This means C recognizes lowercase and uppercase letters, so variables named **aVariable**, **avariable**, and **AVARIABLE** are considered three separate and distinct items.

Capitalization Convention	Examples
All lowercase	interestrate paymentamt lastamtdue
First letter capitalized	Interestrate Paymentamt Lastamtdue
First letter of each word capitalized	InterestRate PaymentAmt LastAmtDue
Words separated with underscores	interest_rate payment_amt last_amt_due

All the examples in the foregoing table are acceptable, but the last two formats (first letter of each word capitalized, and words separated with underscores) are more readable, and are seen most often in the code of professional programmers. Choose one of these styles, or combine them with ideas of your own. Once again, remember to be consistent.

Tip: You may notice that the table of variable-naming examples includes no entries for variables in all uppercase letters. Most C programmers avoid this convention be-

cause it is more difficult to read. Typically, names in all capital letters are limited to #define statements (you will learn more about these in Chapters 8 and 11).

Another popular capitalization convention used in several advanced programming environments (such as Microsoft Windows) is called *Hungarian Notation*. The full definition of this format is beyond the scope of this book. For the purpose of this discussion, however, note in the following examples that variable names in Hungarian Notation are preceded by a lowercase abbreviation of the variable type. The rest of the variable name uses mixed upper- and lowercase letters for readability. When you read code samples from the latest programming books and magazines, you may see examples of this style. In general, it is not recommended for use by the beginning C programmer.

Variable Name in Hungarian Notation	What It Might Mean
iLoanTerm	Integer; Loan Term in years
pzLastName	Pointer to zero-terminated string; Last Name character string
fPayment	Floating point; Payment amount

THE PROGRAMMING PROCESS

Writing a program involves much more than typing in code and watching it run. The process includes planning the program, modularizing your code, writing and testing the program, and documenting it. Following a good design process allows you to write the program once. Neglecting the design process usually means that the program you write won't do what you expect it to, and you'll have to revise it. Examples of an effective design process are shown in Figures 2-1 through 2-3.

The following steps are suggested as the basis for the development of any nontrivial program (more than a few pages of code):

1. **Determine the program's purpose.** Think of all the options that will be required: What will the inputs be? What processing is involved?

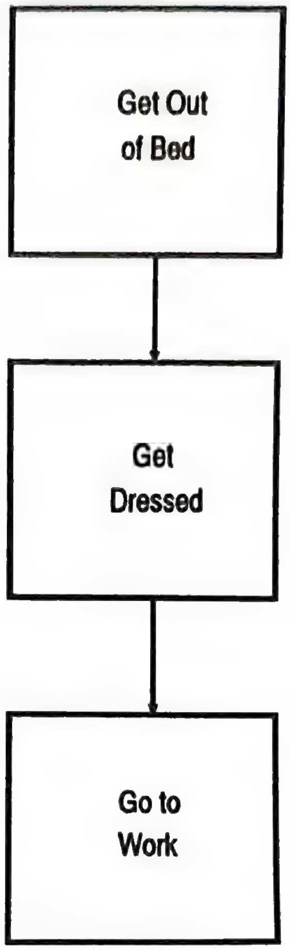


FIGURE 2-1
A simple flowchart

What outputs will be required? For example, a program to calculate loan payments will require inputs for the length of the loan, amount of the loan, and interest rate; processing to calculate the loan payment amount; and printed output of a report on the information input and processing done, along with the final result.

2. **Write down the decisions made in step 1.** If it is a small program for your own use, just put this information in the comments at the beginning of the file. If the program is for someone else, write this information down so the user can agree that the purpose is correct.

   This may seem like overkill, since you've already analyzed the program in the first step. However, writing out the design of the

program, in good detail, often helps you to work out any small problems before it is too late. It also shows your client (whether it is yourself or someone else) exactly what the program will do, before the code is written. This is the best time to locate and correct errors and discrepancies; it also gives you a reference during your development, so you always know what the final program requires.

3. **Decide what functions are needed** so that separate program functions can be written to handle each operation. This includes input functions, functions to perform each of the processing steps, and output (or reporting) functions.
4. **Design the connections between the main program and any separate functions.** A flowchart or pseudocode will help you to create a "roadmap" of the final program (see Figures 2-1 through 2-3 for example flowcharts and pseudocode).
5. **Write the code in small modules, testing each one as it is finished.** If the program is large, group related functions together into their own separate files.
6. **Put the modules together and try the program out as a complete unit.** Debug any problems that arise.
7. **Determine if the program fulfills the original plans.** If so, pat yourself on the back and go on to step 8. If not, return to step one of the design process and find the places where changes need to be made to bring the program into agreement with the original plan.
8. **Complete the documentation.** Even if this program is not for anyone but yourself, always document how the program was designed. This lets you continue to maintain it (that is, correct any bugs and/or make future modifications). The documentation should include all the flowcharts and pseudocode you developed during the programming process.
9. **Take a break,** and *then* start the next project. This pause allows you time to learn from each project.

There are many good books available today that teach and explain, often in great detail, how to handle a programming project; see the Bibliography for a list of some of this author's favorites. The design process outlined above is a combination of several methods, and seems to work best with most projects. Extremely complex projects may require

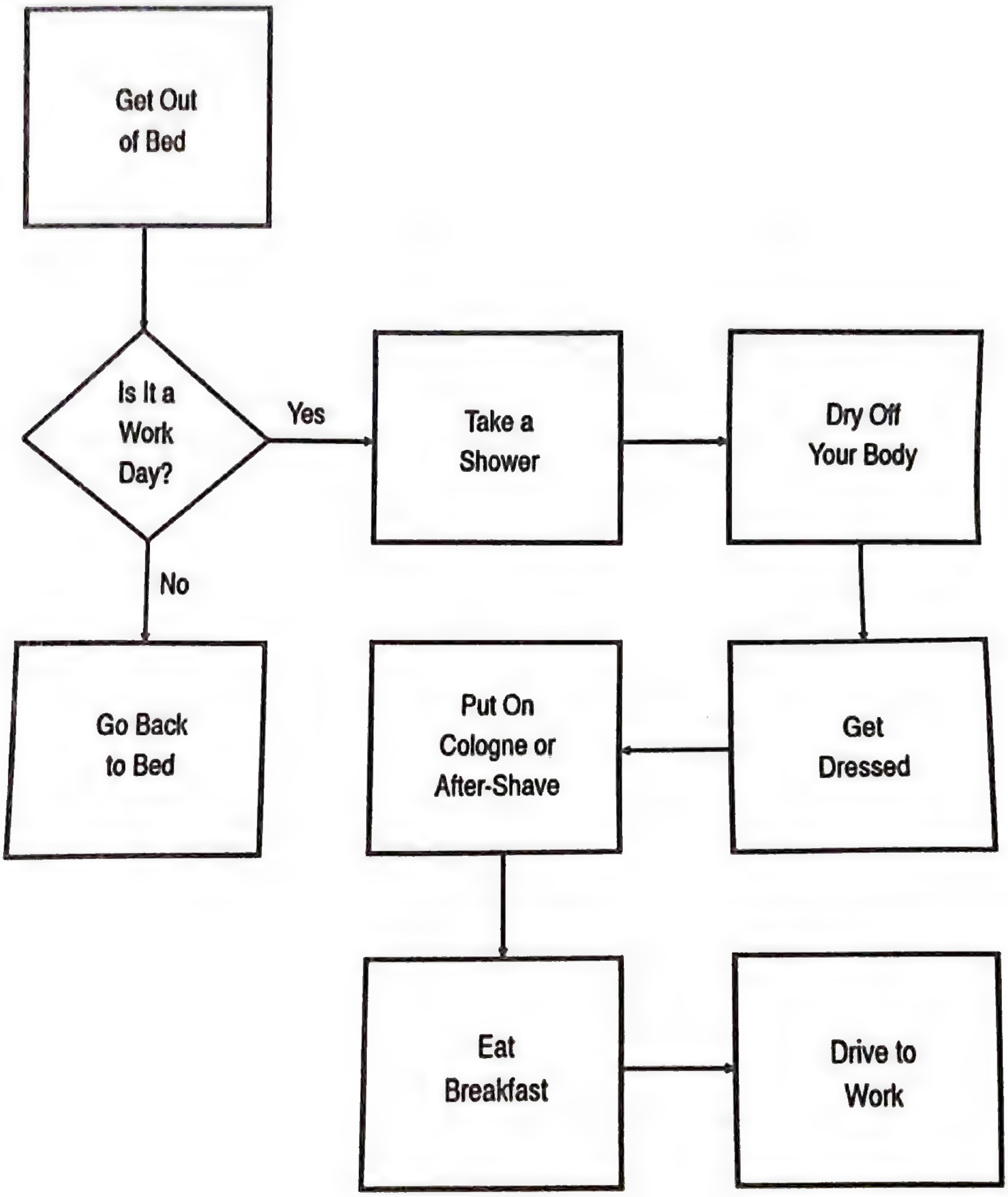


FIGURE 2-2
A more detailed flowchart

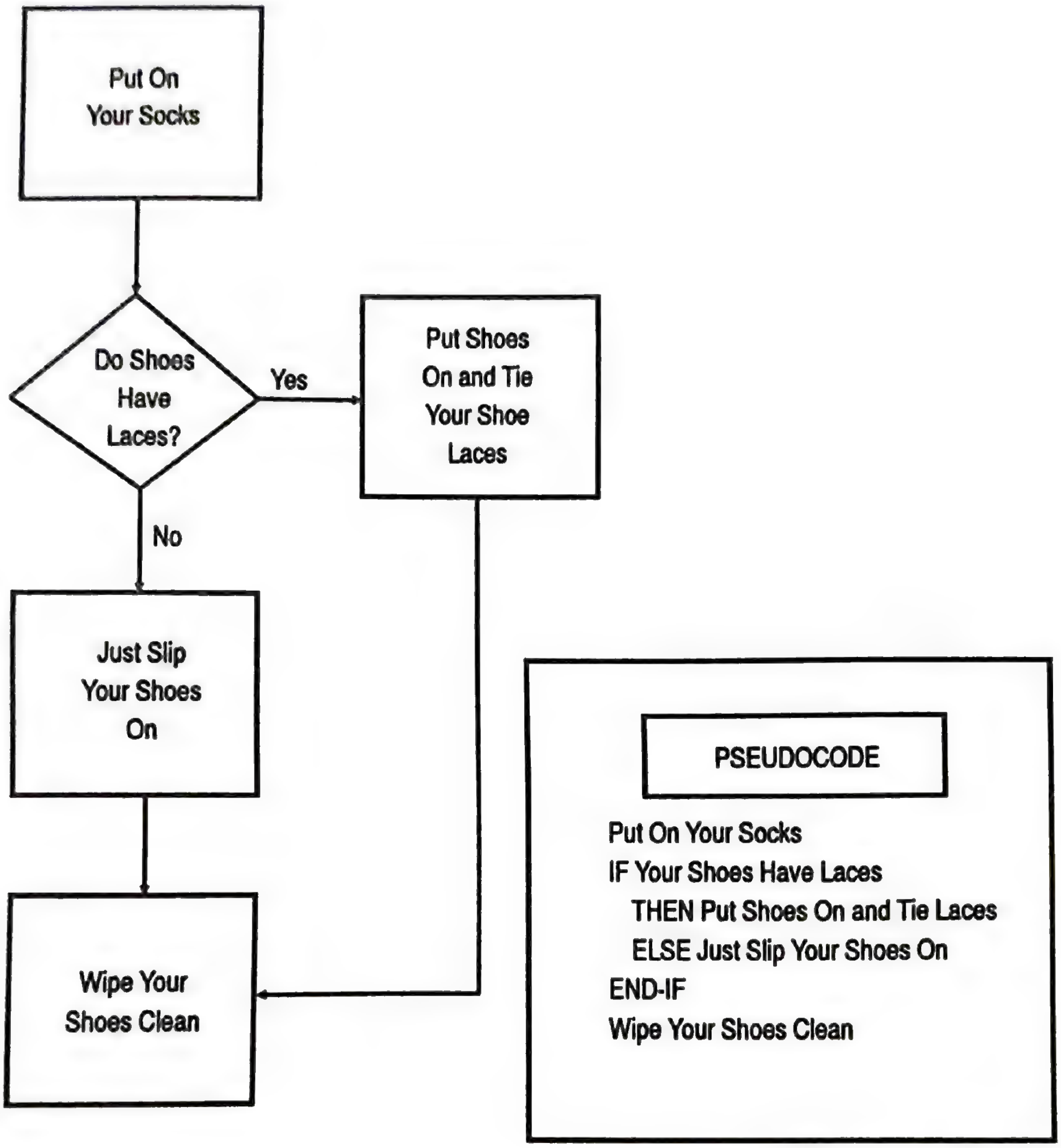


FIGURE 2-3
A flowchart with equivalent pseudocode

additional planning before the actual code is written, and more testing before the program is finished.

Flowcharts are one tool that many programmers use to help design programs, as shown in Figures 2-1 and 2-2. Figure 2-1 is a simple overview of the activities needed to get up and go to work in the morning; Figure 2-2 shows the same sequence of events, in more detail. You can take the same approach with your programming projects: first do an overview diagram, to convey the sequence of events at a superficial level; then work the details of each event into additional diagrams.

Pseudocode is another tool many programmers use in their design tasks. *Pseudocode* is a structured mix of English and C (or another programming language). An example of pseudocode is shown in Figure 2-3, with the equivalent flowchart nearby. Notice how pseudocode may often be simpler to write and follow than a flowchart.

SUMMARY

This chapter showed you the basics of programming style and program design. You learned about

- Indenting blocks of code
- Writing comments
- Naming variables
- The steps in program design and development

The most important point of this chapter, however, is to always be consistent, from program to program, in using the styles you choose for programming in C!

PART TWO

LEARNING TO PROGRAM IN C

C H A P T E R

C PROGRAM FUNDAMENTALS

This chapter has two goals. The first goal is to give you a basic introduction to how C programs look, before we get more specific in the coming chapters. The second is to teach you to use the C DiskTutor environment for entering and running your first C program.

> Note: Since this chapter is intended as an overview of C basics, don't worry about learning all the concepts in one sitting. Just concentrate on getting familiar with the basic principles, so you can recognize and use them in subsequent, more detailed chapters.

C can be both a very simple and a very difficult language to learn. The simplest C program can be very small, such as the one following, Program P3-1.

```
main() {}
```

P3-1

If you ran this program, you would see that it doesn't do anything; but it is a valid C program. Before going on, let's examine the parts of this simple program.

Each function in C has a name (like **main** in Program P3-1), two parentheses, and two curly braces. If parameters will be sent into the function, they are listed within the parentheses. The actual program code for the function is placed between the braces. The details of these portions of a function are covered in the sections that follow.

GETTING STARTED IN C

C programs consist of one or more functions, each of which performs a sequence of events. A *function* is a group or sequence of C statements that are executed together. Every C program begins with a function called **main( )**. This is where program execution begins, and **main( )** is therefore required in every C program.

Unlike many other programming languages, such as Pascal, in which the main routine usually has the same name as the program file, C requires all programs to start with **main( )**. There are no requirements, however, for the filename used to hold the **main( )** function; it can be any valid DOS filename, usually ending with the C extension, such as MYPROG.C or PAYMENT.C.

Note: The name of a C function is usually shown with the parentheses. This tells you the name is a function name, as opposed to a variable name. So the main function in a C program is **main( )**, not main.

Note in Program P3-1 that two parentheses follow **main**. These show that this is a C function. That means that **main( )**, like other C functions, can be called from another function. The parentheses are empty in this case,

but the function's parameters, if used, are placed inside the parentheses. *Parameters* are variables or values sent to the function and used for processing.

The **main( )** function can take an optional set of parameters. These parameters are used for reading the DOS command line and environment (discussed in Chapter 7). Program P3-2, shown below, is an example of a program with parameters for the **main( )** function.

P3-2

```
main ( int argc, char *argv[] )    Parameters
{          Function name                          Comment
   /* Remember that the formatting of your programs
      is your responsibility-a matter of style. This program
      could have been written with the two braces on the
      first line, as was done in Program P3-1. The style
      shown here fits better with the style used throughout
      this book.

      And, of course, this comment is optional!
   */
}
```

In Program P3-2, the first parameter is defined with **int argc**. This declares **argc** to be an integer variable.

The parameter char ***argv[ ]** is a bit more complex; these types of constructs are discussed in Chapter 4. For now, you should know that this parameter is a declaration of **argv** as an array (shown with the square brackets [ ]) of pointers (denoted by the *) to characters; in simpler terms, this parameter is an array of character strings.

Notice that the braces—{ and }—have no code between them. Normally, the actual program code would go inside those braces.

Take a look at another simple program, Program P3-3; this one actually does something. In this section, you will type, or load, Program 3-3 into the DiskTutor environment; then you will get to see what happens when you compile and run it.

P3-3

```
/* A small C program to print a message to the screen. */
/* Written by L. John Ribar in the C DiskTutor. */

#include <stdio.h>

main()
{
```

```
    printf("Hello again! \n");
}
```

Bring up the DiskTutor environment by entering **DTE** at the command line. Your screen will look like Figure 3-1. Then load Program P3-3, by selecting File and then Open from the menus. Or you can start a new file, by selecting File and then New from the menus, and type the program into the window. Either way, your screen should look similar to Figure 3-2 when you finish.

The following paragraphs give a step-by-step explanation of what the program will do when you run it, and how the C language performs each of the required functions.

```
/* A small program to print a message to the screen. */
/* Written by L. John Ribar in the C DiskTutor. */
```

These two lines are comments telling what the program is and who wrote it. Though they are shown as two comments here, this text could have been combined as a single comment like the following one, to reduce typing:

```
/* A small program to print a message to the screen.
   Written by L. John Ribar in the C DiskTutor. */
```

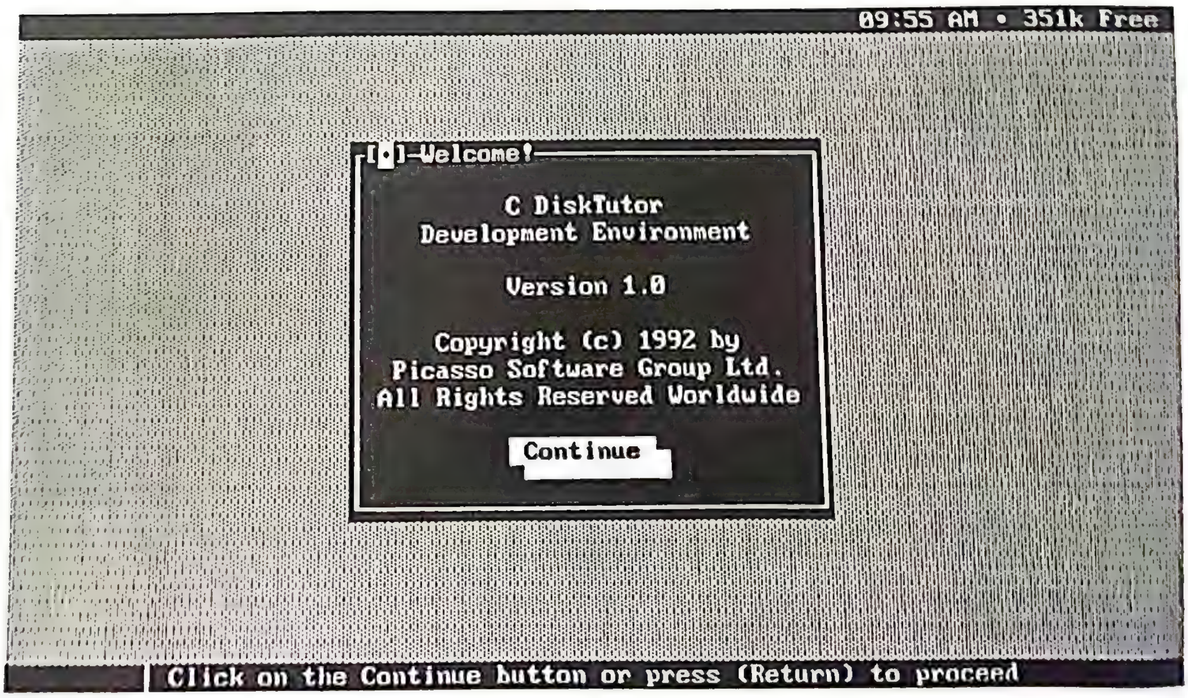


FIGURE 3-1
Opening screen of the C DiskTutor environment

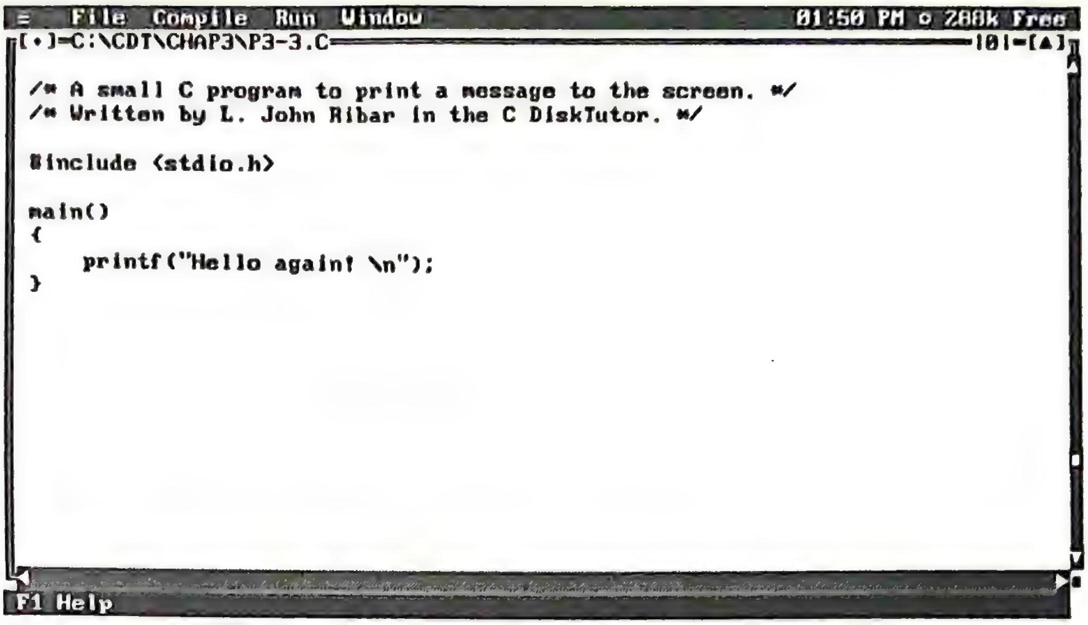


FIGURE 3-2
Your first C program in the DiskTutor environment

```
#include <stdio.h>
```

This line tells the compiler to read the file STDIO.H, and to include its contents in this file.

STDIO.H is a file known as a *header file;* it stores in one place the definitions used in many programs, so the definitions don't have to be retyped for each program. The STDIO.H header file contains the information about input and output functions. See Chapter 8 and Chapter 11 for a more complete discussion.

The <> delimiters around the STDIO.H filename show that the file is a system file. System files come with the compiler, and are stored in a special directory that the compiler knows about and can find. If the filename were surrounded by quotation marks instead, like this:

```
#include "stdio.h"
```

then the compiler would search for the header file in the current directory rather than the system directory. Your own header files will generally be included by using the quotation mark (or local) version. A *local* file is a file found in the current subdirectory.

```
main()
```

This is the start of the main program. As explained earlier in this chapter, the **main( )** function can be declared with or without parameters, which you can either use or ignore. Functions other than **main( )** either will always use parameters, or will not ever have any parameters.

```
{
```

The first brace marks the beginning of the code of the **main( )** function. (Notice that both this brace and the one at the end of the program are in column one, and that this block of code is indented. This format is consistent with the style adopted in Chapter 2 for use throughout this book. See Chapter 2 if you need to review program indentation and other style issues.)

```
printf("Hello again! \n");
```

This line contains a call to the function named **printf( )**. In this case, a parameter—the string **Hello again! \n**—is sent into the function by placing it between the parentheses. The function **printf( )** is a very common one, and you will see it used throughout the book; it is fully explained in Chapter 9.

The **\n** at the end of the string is a shortcut notation for a carriage return, which makes the cursor move to the beginning of the next line on the screen, after the string is printed. This is like pressing the carriage return on a typewriter after typing a line.

```
}
```

This brace marks the end of the **main( )** function and, in this case, the end of the program, too.

Note: In C, braces must always "match up" in pairs; every opening brace ({) must have a matching closing brace (}). If the compiler finds an unmatched brace, it will flag it as an error, and you will need to supply the missing (or delete the extra) brace.

Once you have entered Program P3-3 into a window, save it using File and Save from the menus with the name HELLO.C, and exit the DiskTutor environment. Next you will use the DiskTutor C compiler to compile the

program. To do this from the DOS command line, enter the following command:

```
WCL hello.c
```

To run the program from the DOS command line, simply type the filename and press ENTER. For instance, if you have named the program HELLO.C, type HELLO. After compiling and running this program, your screen will look similar to Figure 3-3. Or, to compile and run the HELLO.C program from within the DiskTutor environment, type (or load) the program as shown above, press ALT-C to compile it, and ALT-R to run it. When calling the compiler and running the program from within the DiskTutor environment, your screen will also look similar to Figure 3-3.

CHARACTERS USED IN C

The C language uses a standard set of characters. Digits are used to represent numbers; alphabetic characters, digits, and the underscore character (_) can all be used in variable names. Any character can be part of a

```
C:\CDT>wcl hello.c
WATCOM C Compile and Link Utility Version 8.5
Copyright by WATCOM Systems Inc. 1988, 1991.  All rights reserved.
WATCOM is a trademark of WATCOM Systems Inc.
       wcc hello.c
WATCOM C Optimizing Compiler  Version 8.5c
Copyright by WATCOM Systems Inc. 1984, 1991.  All rights reserved.
WATCOM is a trademark of WATCOM Systems Inc.
hello.c: 10 lines, included 155, 0 warnings, 0 errors
Code size: 17

WATCOM Linker Version 7.0
Copyright by WATCOM Systems Inc. 1985, 1991.  All rights reserved.
WATCOM is a trademark of WATCOM Systems Inc.
loading object files
searching libraries
creating a DOS executable

C:\CDT>hello
Hello again!

C:\CDT>
```

FIGURE 3-3

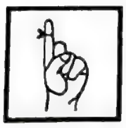
Your first compile and run from the command line

character string. Most mathematical functions are performed using the same characters seen on calculators.

You have read in Chapter 2 that variables are storage areas for C programs (you'll learn more about data storage in Chapter 4). You also saw some examples of variable names in Chapter 2. All variable names must start with either a letter or an underscore character; the first character can then be followed by any number of letters, digits, or underscores. You cannot use spaces or other characters (such as question marks, exclamation points, pound signs, and so on) within variable names.

Some valid and invalid variable names and their limitations are listed here:

Valid Variable Name	Invalid Variable Name	Limitation
myVariable	my variable	Spaces not allowed
room4	4room	Starts with a number instead of a letter or underscore
last_name	AreYouGoing?	? is an illegal character
_okName	ok-name	- is an illegal character



Remember: C is a case-sensitive language. Thus myVariable, myvariable, and MYVARIABLE represent three different variables.

Some compilers limit the maximum length of variable names, often to 32 characters. The DiskTutor compiler allows variable names of any length, but recognizes only the first 40 characters; the rest of the characters are ignored.

Reserved Words The words in the following list are called *reserved words*. Reserved words all have special meaning within the C language, and the compiler will therefore not allow their use for naming variables or functions.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static

struct	switch	typedef	union
unsigned	void	volatile	while

Check the manual for your compiler to find other words that may also be reserved for use by that compiler. For instance, the DiskTutor and many other compilers reserve these words:

cdecl	far
fortran	huge
near	pascal

SPECIAL CHARACTERS

The nonalphabetic and nonnumeric characters listed in Table 3-1 are part of C. While covered in more depth as you progress through the book, this list serves as a quick reference.

Characters	Usage
{ }	Delimits blocks of code
( )	Delimits the parameter list in function definitions or function calls
[ ]	Delimits the index of an array variable
< >	Delimits the name of a standard header file
>	Greater-than comparison in equations
<	Less-than comparison in equations
==	Is-equal-to comparison in equations
>=	Greater-than-or-equal-to comparison in equations
<=	Less-than-or-equal-to comparison in equations
!=	Is not equal to
!	NOT; negates value of item to its right
~	Complement of a number
/*	Begins a comment
*/	Ends a comment
' '	Delimits a single character
" "	Delimits a string of characters; also used to delimit name of a local header file instead of < and >

TABLE 3-1
Nonalphabetic and Nonnumeric Characters Used in C

Characters	Usage
*	Serves as a multiplication symbol in equations; or to denote either a pointer variable or the contents of a pointer variable
+	Addition symbol in equations
-	Subtraction symbol in equations
/	Division symbol in equations
%	Modulus symbol in equations; similar to division, but returns the remainder after dividing two items
+	Increments a number by one
--	Decrements a number by one
|	ORs two numbers together
||	ORs the boolean value of two expressions together
&	Denotes the address of the variable to its right; or ANDs two numbers together
&&	ANDs the boolean value of two expressions together
>>	Arithmetic shift-right function
<<	Arithmetic shift-left function
^	Represents EXCLUSIVE-OR of two values
?	In special C statements, determines which of two values to choose, based on a boolean operation
:	Separates choices used in conjunction with ? operation
#	Introduces a preprocessor directive
\	Introduces a special character in a character constant, or as part of a character string
;	Ends every C statement
=	Assigns values to a variable
+=	Adds value on right to variable on left; combination of addition and assignment functions
-=	Subtracts value on right from variable on left; combination of subtraction and assignment functions
*=	Multiplies value on right to variable on left; combination of multiplication and assignment functions
/=	Divides variable on left by value on right; combination of division and assignment functions
^=	EXCLUSIVE-ORs value on right and variable on left; combination of EXCLUSIVE-OR and assignment functions

TABLE 3-1

Nonalphabetic and Nonnumeric Characters Used in C (continued)

From Pascal to C

Throughout this book, comparisons will be made between C and Pascal. If you know Pascal, you'll want to read these boxed sections—they'll help you learn C by equating it to Pascal. At this point in the book, you may not think C shares many similarities with Pascal. Here are two programs, one in C, Program P3-4, and one in Pascal, Program P3-5, that might help you begin to see the similarity:

P3-4

```
/* A C demo program */
#include <stdio.h>    /* include input/output routines */

main()                Declaring a variable
{
   int i;     /* declare an integer variable */
   i = 1;     /* assign a value of 1 to i */
   if (i==1)                                   <-- An if statement
      printf("i is equal to one \n");
   else
   {  /* '{' and '}' surround a block of code */   A block of code
      printf("i is not equal to 1 \n");
      i = 1;
   }
}
```

P3-5

```
(* a Turbo Pascal demo program *)

program PasDemo;

uses CRT;    (* include input/output routines *)

VAR              Declaring a variable
   i : INTEGER;   (* declare an integer variable *)
BEGIN
   i := 1;    (* assign a value of 1 to i *)
   IF i = 1 THEN                               <-- An IF statement
      WriteLn('i is equal to 1')
   ELSE
   BEGIN  (* BEGIN and END surround a block of code *)   A block of code
      WriteLn('i is not equal to 1');
      i := 1;
   END;
END.
```

From Pascal to C (*continued*)

Though these are trivial examples, and you may not yet understand everything that is happening in the C program, notice these similarities: how variables are declared, the use of **if-else** statements, and the blocks of code. They will be explored in subsequent chapters.

Remember: Unlike many other languages, such as Pascal and Fortran, C is case sensitive!

SUMMARY

This chapter introduced you to the very basics of C programming, and showed you how to compile and run your first C program. You learned

- The parts of a small C program, including comments, variables, and functions
- The rules for naming variables
- The reserved words of C
- The special characters used in C

Now, you are ready to learn more about the art of programming in C.

C H A P T E R

DATA STORAGE AND MANIPULATION

Before you begin to write programs in C, or in any computer language, you need to have a way to store the information that you will manipulate. This information can appear in many forms, so the C language has many different ways to handle it.

This chapter deals with the first round of data storage concepts in C. In the last chapter, you learned how to name variables. Now, you will learn the variable types, and how to manipulate them in a program. These are the foundations upon which you will build a C program.

VARIABLE TYPES

All programs are written to manipulate data in some form; therefore, how data is stored is very important in almost all computer programs. Naturally, if your only purpose in writing a program is to display your name, data storage is not needed. Nor does a program that adds two numbers and prints the results, as shown in Program P4-1, have much need of data storage.

P4-1

```
/*
   A basic C program with no variables.
   By:  L. John Ribar, C DiskTutor
*/

main()
{
   printf("The sum of 4 plus 5 is %d \n", 4+5 );
}
```

A program that displays the result of adding two numbers is only useful if the numbers can be changed without rewriting the program. To accomplish this, the two numbers are stored in variables, which are then added. Each time the program is run, you enter these variables and voila!, you have created a simple calculator. Program P4-2 shows how a simple calculator program might look.

P4-2

```
/*
   A simple calculator program.
   This program allows the user to enter two numbers, and
   then prints the result of adding those numbers.
   By:  L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i;      /* the first variable */
   int j;      /* the second variable */
   int total;  /* the total value */

   printf("What is the first number? ");
   scanf("%d",&i);

   printf("What is the second number? ");
   scanf("%d",&j);

   total = i + j;
   printf("The total is %d \n",total);
}
```

Load this program into the DiskTutor environment now, and study the following explanation of how the program works.

The **#include** statement at the top of the program tells the compiler that the stdio.h header file should be included in this program. This file contains descriptions of several of the more common input and output features, including **printf( )** and **scanf( )**, used in this example.

> Tip: In an **#include** statement, if the filename is surrounded by < and >, as in <stdio.h>, the program searches for the file in the system header file directory. If surrounded by " and ", as in "stdio.h", the file will be located in the local (current) directory.

Three lines in Program P4-2 begin with the word **int**. These are the *declarations* for our variables; **int** means that these are *integer* values. Notice that variables are declared at the top of the **main( )** function. After these declarations, actual lines of code begin, and no more declarations of variables may be made within that function.

For the sake of simplicity, let's agree that the **printf( )** function displays text on the screen, and the **scanf( )** function reads characters from the keyboard (these functions will be explained in detail in Chapter 9). For now, remember that both functions take a *format string* (in quotes) and zero or more additional parameters (variables or values). Within the format string, **%d** is used to represent a decimal value. The value to put in that position is one of the parameters listed at the end of the function call.

Finally, notice the line that looks like basic mathematics—the line where **total** is assigned the value of i plus j.

This is all the work it takes to create a simple addition-only calculator. Run the program now, and see how the results are displayed. Your screen will show results similar to this:

```
C:\CDT>p4-2
What is the first number? 10
What is the second number? 20
The total is 30

C:\CDT>
```

Before you look at any more code, let's explore different variable types, such as how the **int** variables in Program P4-2 are declared and used.

INTEGERS

The integer is one of the most basic of variable types used in C programs. An *integer* is simply a number that has no fractional part, or that is a counting number. For instance, the following numbers are valid integers:

```
5
125
-27
32100
```

and the following are not integers:

```
5.24
3/4
-32.112
AQJ-317
```

There are many types of integer variables available in C, with the main difference between them being the range of numbers covered. Let's examine the standard types of integers, and examples of their declaration and use.

Integer Declarations

In order to have increased storage capability, or to save space, integers are declared in several ways. The basic way to declare an integer is

```
int i;
```

In standard C (now referred to as ANSI C), the **int** variable type can handle a range of values from -32,767 to 32,767. This variable type is often used for counting tasks needed in C programs.

Note: The ranges given for different variable types are from the ANSI (American National Standards Institute) C standard. Your compiler may have a different range based on the machine for which the compiler is written. The ranges given are the minimum ranges allowed, so they will always be available from an ANSI C compiler. If your program uses values close to the limits, check your compiler for its particular restrictions.

To save space in your programs, you can use the following declarations:

```
short int i;
short j;
```

The **short** designator, with or without the optional **int** keyword, defines a variable with a range from –32,767 to 32,767. A **short int** is used to store a small integer number, and is always shorter than or equal to the size of an **int**. In many PC compilers, **int** and **short int** variables are identical in size and range. This is not always actually true, however, as compilers and machines may differ in how the sizes are defined. On some computers, a **short int** will be eight bits long, and an **int** will be sixteen bits, while another computer's compiler may define both types as sixteen bits.

In handling larger integer numbers, a **long** declaration is used, like this:

```
long int i;
long j;
```

Using the **long** designator, again with or without the optional **int** keyword, opens the range of this variable type from –2,147,483,647 to 2,147,483,647. The **long** declaration uses twice the storage of an **int** variable. Again, although this range is defined by the ANSI standard, the compiler you use may extend the range based on your hardware.

VARIABLES WITH ONLY POSITIVE VALUES

You may have noticed that the variable types mentioned thus far have all included negative and positive numbers. Variables that contain only positive numbers are declared by placing the keyword **unsigned** before the normal declaration, like this:

```
unsigned int i;
unsigned short int j;
unsigned long int k;
```

In using only positive numbers, the range of each **unsigned** integer is doubled. That is, a normal **int** variable has a range of –32,767 to 32,767, but an **unsigned int** variable has a range of 0 to 65,535, because no values can be negative.

What happens if the number is out of the variable's range? Load the following, Program P4-3, into the DiskTutor environment, and run it.

P4-3

```
/*
   A C program with an error. This shows what the compiler
   will do with out-of-range values.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i;  /* declare the integer variable */

   i = 100000;  /* assign a number that is out of range */
   printf("The value of i is %d \n", i );
}
```

When you run the program, the results will look similar to those shown below. Notice the output—it's probably not what you would expect. This error is caused by out-of-range values.

```
C:\CDT>wcl p4-3
WATCOM C Compile and Link Utility Version 8.5
Copyright by WATCOM Systems Inc. 1988, 1991. All rights reserved.
WATCOM is a trademark of WATCOM Systems Inc.
        wcc p4-3.c
WATCOM C Optimizing Compiler Version 8.5c
Copyright by WATCOM Systems Inc. 1984, 1991. All rights reserved.
WATCOM is a trademark of WATCOM Systems Inc.
p4-3.c(13): Warning! W106: Constant out of range; truncated
p4-3.c: 15 lines, included 155, 1 warnings, 0 errors
Code size: 21
```

Notice that a warning is generated by the compiler.

```
WATCOM Linker Version 7.0
Copyright by WATCOM Systems Inc. 1985, 1991. All rights reserved.
WATCOM is a trademark of WATCOM Systems Inc.
loading object files
searching libraries
creating a DOS executable

C:\CDT>p4-3
The value of i is -31072

C:\CDT>
```

SIMPLE MATHEMATICAL OPERATIONS

Using integer values in a program is fairly simple, and all the standard mathematical operations you use for numbers are available for integers. Some of the functions that can be performed on integers are shown below; in this table, assume that i1, i2, and i3 are declared as int variables, although short int and long int variables will also work.

Operation	C Example
Assignment	i3 = 10;
Addition	i3 = i1 + i2;
Subtraction	i3 = i1 – i2;
Multiplication	i3 = i1 * i2;
Division	i3 = i1 / i2;
Remainder (or Modulus)	i3 = i1 % i2; (i3 will be assigned the remainder of the division of i1 and i2)

Some simple mathematical operations, using the different integer types as examples, are shown in Program P4-4.

P4-4

```
/*
   C program to demonstrate the use of integers.
   By: L. John Ribar, C DiskTutor
*/

#include <stdio.h>

main()
{
   /* Declare 3 integers. They can be declared all at one
      time by separating their names with commas, as shown
      here. */
   int i1, i2, i3;

   /* Now declare 3 short integers, on three separate lines.
      Again, these could have been declared all on the same
      line by separating their names with commas, as done
      above. */
   short int s1;
   short int s2;
   short int s3;
```

```
    /* Finally, declare 3 long integers. */
    long int L1, L2, L3;

    /* Assign values and display int values */
    i1 = 10250;        /* assign a value to i1 */
    i2 = 50;           /* a value for i2 */
    i3 = i1 + i2;      /* add them together */
    printf(" i1 = %d  and  i2 = %d, i1 + i2 = %d \n",
       i1, i2, i3);    /* print out the values */

    /* Now work with the short values */
    s1 = 25;
    s2 = 5;
    s3 = s1 / s2;
    printf(" s1 = %d  and  s2 = %d, s1 / s2 = %d \n",
       s1, s2, s3);    /* print out the values */

    /* Long integer examples */
    L1 = 230040;
    L2 = 750;
    L3 = L1 % L2;
    printf(" L1 = %ld  and  L2 = %ld, L1 %% L2 = %ld \n",
       L1, L2, L3);    /* print out the values */
                       /* notice that '%%' was used to print
                          the modulus operator, so that the
                          compiler would not use the '%' as a
                          format specifier. */

    /* now something a little more fun */
    L1 = 3;
    L2 = 15;
    L3 = (10*L1) + (L2/5);
    /* print out the values */
    printf(" L1 = %ld  and  L2 = %ld,",L1,L2);
    printf(" (10*L1)+(L2/5) =  %ld \n", L3);
    /* Note that multiple printf() lines can be used to
       complete a statement. The first printf() does not have
       a '\n', so the second printf() will continue printing
       on the same line. */
}
```

Note: The integer variables respond to the concept of parentheses in an equation. That is, the expressions within the parentheses are performed first, with the result passed into the outer expressions. C uses the standard rules of precedence for mathematical operations. Expressions are evaluated from left to right, multiplication and division are performed first, and addition and subtraction second. Parentheses are used to reorder the evaluation.

Compile Program P4-4 now and execute it. The output from using integer variables is shown here:

```
C:\CDT>p4-4
 i1 = 10250 and i2 - 50, i1 + i2 = 10300
 s1 = 25 and s2 = 5, s1 / s2 = 5
 L1 = 230040 and L2 = 750, L1 % L2 = 540
 L1 = 3 and L2 = 15, (10*L1)+(L2/5) = 33

C:\CDT>
```

Notice in Program P4-4 that the formatting in the **printf( )** statement changes with the different types of variables; this is also true with **scanf( )**. The codes in the format portion of the **printf( )** and **scanf( )** statements that begin with the percent sign (%) are called *format specifiers*. Here are the format specifiers used with integer variables:

Format Specifier	Variable Type
%d	int
%i	int
%sd	short int
%si	short int
%ld	long int
%li	long int
%u	unsigned int
%hu	unsigned short int

Format Specifier	Variable Type
%lu	unsigned long int
%o	int printed as an octal value
%ho	short int printed as an octal value
%lo	long int printed as an octal value
%x	int printed as a hexadecimal value
%hx	short int printed as a hexadecimal value
%lx	long int printed as a hexadecimal value
%X	int printed as a hexadecimal value
%hX	short int printed as a hexadecimal value
%lX	long int printed as a hexadecimal value

ADVANCED OPERATIONS

Although the basic integer math expressions shown so far will allow you to write a powerful number manipulation program, C provides a rich set of advanced operations for use with integer variables. These additional features make your programs simpler to read and debug.

Note: Debugging a program is the process of finding errors in it. Making the program simple to read saves time and promotes understanding of the program's purpose when you're debugging.

Auto-increment and Auto-decrement

In C, a variable can be incremented (add 1 to the variable) or decremented (subtract 1 from the variable) using the *auto-increment* and *auto-decrement* operators. Suppose you were to write a simple expression to add 1 to a variable x, knowing what you have already learned:

```
x = x + 1;
```

That was not so difficult. Now, suppose that the name of the variable you wish to increment is

```
TheLongestNameOfAVariableThatYouMightEverWant
```

Adding 1 to this variable would look like this:

```
TheLongestNameOfAVariableThatYouMightEverWant =
TheLongestNameOfAVariableThatYouMightEverWant + 1;
```

Of course, you might never name a variable with a name that long. Whether you do or not, C has a way to add 1 to the value of a variable that is much simpler than what you see above. Just place the auto-increment operator ++ before or after the variable name (either way is fine). Subtracting 1 from the value of a variable is just as easy; place the auto-decrement operator -- before or after the variable.

```
i++;   /* Add 1 */
++i;

i--;   /* Subtract 1 */
--i;
```

Whether you put the ++ or -- before or after the variable depends on the outcome you need. If your only purpose is to increment or decrement the variable, you can place the operator in either location. However, increment and decrement operations can also be used within other statements. In these cases, the position of the operator *does* make a difference. When the operator is in front of the variable, the increment or decrement is performed before the variable is used.

In this example,

```
j = ++i;
```

i is incremented before its value is assigned to j. If i had a value of 5 before this statement, it would be incremented to 6, and then j would get the value 6. On the other hand, in this next example,

```
j = i++;
```

if i had a value of 5, j would first get the value 5, and then i would be incremented to 6.

Program P4-5 shows a program that performs several increment and decrement operations. Load and run it now to see how the auto-increment and auto-decrement operators can be used.

P4-5

```
/*
   C program to test auto-increment and auto-decrement
   operations on integers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   /* declare two integer variables */
   int i, j;

   i = 1;       /* initialize i to 1 */
   j = i++;     /* i = 2, j = 1 because i was incremented
                   after the assignment of i to j */
   printf(" i = %d, j = %d \n", i, j );

   j = ++i;     /* After the statement,
                i is 3 because of the increment,
                and j is also 3 because the increment
                happened before the assignment. */
   printf(" i = %d, j = %d \n", i, j );

   j = 3*(i--);
                /* Before this assignment, i and j are both
                3. After this statement executes,
                i is 2, and j is now 9. i is 3
                until after the assignment is
                complete, because the -- follows i. */
   printf(" i = %d, j = %d \n", i, j );
}
```

Here is the output from Program P4-5:

```
C:\CDT>p4-5
 i = 2, j = 1
 i = 3, j = 3
 i = 2, j = 9

C:\CDT>
```

You can see that placing the auto-increment or auto-decrement operator before the variable causes the action to take place before the variable is used, and placing the operator after the variable causes the variable to be used before the action is taken.

The auto-increment and auto-decrement operators can also be used in a simple statement for the addition of 1 to a variable. For instance, using a previous example, you can now use

```
TheLongestNameOfAVariableThatYouMightEverWant++;
```

In this case, it makes no difference whether the operator is before or after the variable name. The statement simply adds one to your variable.

Shift Left and Shift Right

Shifting a variable moves the actual bits within the integer, in one direction or the other. For instance, the number 10 is represented in binary as 00001010. If you perform a *shift right*, all the bits are moved to the right, resulting in 00000101. Notice that this is the binary representation of the value 5, or one-half of the original value. This works in reverse with a *shift left*; ten (00001010) becomes twenty (00010100), twice the original.

This phenomenon occurs because the computer uses binary arithmetic internally. Therefore, a shift in either direction will change the original value by a factor of two. Very often, the shift operation is used to multiply and divide by powers of two. Since two to the second power is four, multiplying a number by four involves a simple shift left by two bits. Ten (00001010) becomes forty (00101000); ten times four is definitely forty!

The shift operations are performed with the >> operator for shifting right, and the << operator for shifting left. The shift operator is followed by a number denoting the number of bit positions to move. To see an example of this, read through Program P4-6.

Tip: Using shift left and shift right in your programs makes the programs work faster than if you use normal multiplication and division. Unfortunately, these operators only work if you multiply or divide by a power of two (such as 2, 4, 8, 16, 32, and so on).

P4-6

```
/*
   C program to demonstrate bit-shifting.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
```

```
    int i;   /* declare an integer variable */
    int j;   /* another integer variable */

    i = 1;   /* assign an intial value */
    j = i << 2;    /* j will now equal 4 */
    printf(" i = %d, j = %d \n", i, j );

    /* Since shifting just moves bits, shifting left and then
       right again will restore the original value. */
    j = j >> 2;    /* j should be 1 again */
    printf(" i = %d, j = %d \n", i, j );
}
```

Numbering Systems in C

In C, numbers can be represented in several different ways. The most common numbering systems are called *decimal, octal, hexadecimal* (or hex), and *binary*. In these systems a number's representation is determined by the *base* of the number being used. Hence, decimal numbers use base 10, octal numbers use base 8, hex numbers use base 16, and binary numbers use base 2. What does this mean?

A binary number is a series of digits, all 0's and 1's, which as a group represents a number. An octal number representation uses a group of digits between 0 and 7, decimal uses a group of digits from 0 to 9, and hex uses 0 to 9 and A to F. If an **int** variable has a storage length of 16 bits, then it will require 4 hex digits, 6 octal digits, or 16 binary digits to represent the value.

Note: Internally, PCs use 16 bits to store most integers. However, showing sixteen bits for binary numbers is somewhat messy, so the binary examples in this section assume the use of an 8-bit number. Although a 16-bit number would have the same value, it would be displayed with more 0's to the left: 0000000000000101 instead of 00000101.

Figure 4-1 shows you graphically how numbers are represented in each of the numbering systems; and Table 4-1 shows you a comparison of the four types of numbers.

Note: Zero and one are special numbers; they are the same in all bases—binary, octal, decimal, and hexadecimal.

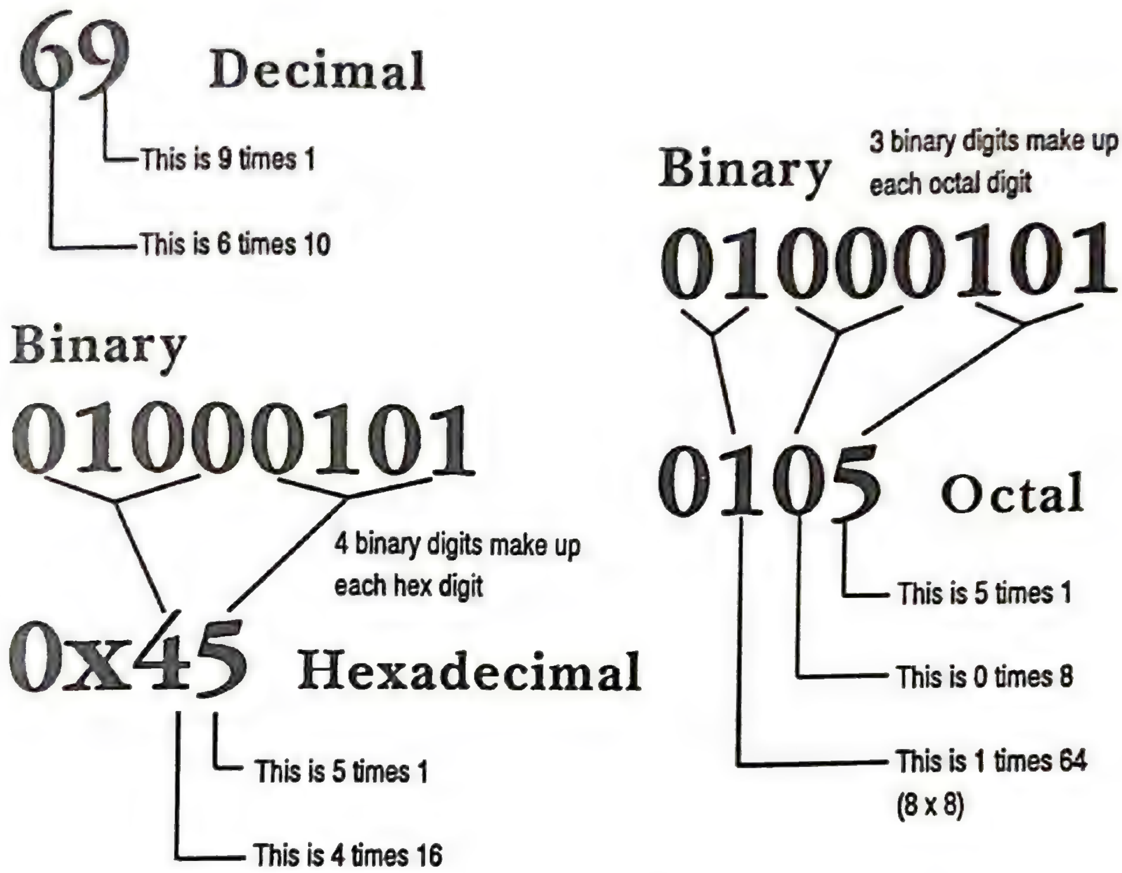


FIGURE 4-1
A comparison of the different representations of the number 69

Although not often done, a number may be used in a base other than ten, if it is correctly written. This method is very helpful in technical applications, where you may be using data that is presented in hex. The first two characters of all hex numbers are 0x or 0X (depending on whether the hex number is shown in lowercase or uppercase letters). A decimal number starts with any nonzero digit, and octal numbers always begin with a zero. These formats are shown in Table 4-1. All the statements shown on the next page make the equivalent assignment.

Decimal	Binary	Octal	Hex
0	0000	000	0x00
1	0001	001	0x01
2	0010	002	0x02
3	0011	003	0x03
4	0100	004	0x04
5	0101	005	0x05
6	0110	006	0x06
7	0111	007	0x07
8	1000	010	0x08
9	1001	011	0x09
10	1010	012	0x0A
11	1011	013	0x0B
12	1100	014	0x0C
13	1101	015	0x0D
14	1110	016	0x0E
15	1111	017	0x0F

TABLE 4-1
A Comparison of Numbers in Different Bases

```
i = 10; /* decimal */
i = 012; /* octal */
i = 0xA; /* hexadecimal */
```

Bit-level Operations

Although shifting a number's bits left and right can be used to provide many answers, C also gives you the ability to actually look at individual bits within a word. The bitwise OR, AND, NOT, and XOR (exclusive or) operations are used to compare and manipulate C variables at the binary, or bit, level. These operations are performed with the | (OR), & (AND), ~ (NOT), and ^ (XOR) symbols. Figure 4-2 shows how the bits are changed using these manipulators.

Bitwise OR The OR operation is used when you want the bits from either of two variables to be available in the result, as shown in Program P4-7.

AND	If first variable is ...	1	1	0	0
	And second variable is ...	1	0	1	0
	Then the result will be ...	1	0	0	0

OR	If first variable is ...	1	1	0	0
	And second variable is ...	1	0	1	0
	Then the result will be ...	1	1	1	0

NOT	If first variable is ...	0	1
	Then the result will be ...	1	0

XOR	If first variable is ...	1	1	0	0
	And second variable is ...	1	0	1	0
	Then the result will be ...	0	1	1	0

FIGURE 4-2
Changing bits using AND, OR, NOT, and XOR

P4-7

```
/*
   C program to test bitwise OR.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i, j, k; /* declare two integers */

   i = 10;      /* 00001010 */
   j = 22;      /* 00010110 */
   k = i | j;   /* 00011110 is the result of OR - put
                            a 1 where either i or j had a 1 */

   printf(" i = %d, j = %d, and k = %d\n", i, j, k );
}
```

Bitwise AND The AND operation gives a result that has only the bits that existed in both the original values, as shown in Program P4-8.

P4-8

```
/*
   C program to test bitwise AND.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i, j ,k;  /* three integer variables */

   i = 10;       /* 00001010 */
   j = 28;       /* 00011100 */
   k = i & j;    /* 00001000 is the result of AND - only the
                             bits where both i and j had a
                             1. */

   printf(" i = %d, j = %d, and k = %d\n", i, j, k );
}
```

Bitwise NOT The NOT operation is used to reverse all the bits in a word, so all 1's become 0's, and all 0's become 1's, as shown in Program P4-9.

P4-9

```
/*
   C program to test bitwise NOT.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i, j;     /* integer variables */

   i = 10;       /* 00001010 */
   j = ~i;       /* 11110101 is the result of NOT - all the
                             1's are turned to 0's, and
                             0's to 1's */

   printf(" i = %d and j = %d\n", i, j );
}
```

Bitwise XOR The XOR (exclusive or) operation produces a result that includes a bit (1) if it exists in one of the original variables, but not in both, as shown in Program P4-10.

P4-10

```
/*
   C program to test bitwise XOR.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i, j, k; /* three integers */

   i = 10;       /* 00001010 */
   j = 28;       /* 00011100 */
   k = i ^ j;    /* 00010110 is the result of XOR - each
                    bit is 1 if only one of the
                    variables had a one. If both were
                    one or both were zero, then the
                    resulting bit is 0 */

   printf(" i = %d, j = %d, and k = %d\n", i, j, k );
}
```

To get a better idea of how the bitwise operations work, compile and run the program file for Program P4-11 several times.

P4-11

```
/*
   C program to demonstrate several bit-level operations.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i, j;  /* integer variables */

   printf("Enter a small number: ");
   scanf("%d",&i);
```

```
    j = i & 0x0F;   /* Pull off the the bottom 4 bits, by
                       ANDing i with 0x0F (00001111) */
    printf("\n The bottom 4 bits equal %d \n",j);

    j = i << 1;     /* shift left */
    printf("i shifted to the left is %d \n",j);

    j = i >> 1;     /* shift right */
    printf("i shifted to the right is %d \n",j);

}
```

Timesavers—Operation with Assignment

There are several shortcuts available in C. Any time a simple operation (such as addition, subtraction, multiplication, division, or modulus) is to be performed on a given variable, an operation with assignment operator may be used. For example,

```
i = i + 5;
```

can be stated more simply as

```
i += 5;
```

These two statements perform the same operation, but the latter one is simpler and requires less typing. It is not required that you use this shorthand notation; do not use it if you feel it will make your code harder to understand.

Other operations are also shortened in the same way, as shown here:

Original Version	Shortcut
i = i + 6;	i += 6;
j = j - 5;	j -= 5;
k = k * 12;	k *= 12;
m = m / 4;	m /= 4;
n = n % 2;	n %= 2;
p = p << 1;	p <<= 1;
q = q >> 4;	q >>= 4;

Original Version	**Shortcut**
r = r | 234;	r |= 234;
s = s & 1022;	s &= 1022;
t = t ^ 91;	t ^= 91;

CHARACTER DECLARATIONS

Characters are actually treated as numbers in C, but numbers with very limited ranges. Character declarations can be any of the following:

Character Declaration	**Numeric Range**
char ch;	0 – 127
signed char ch2;	-127 – +127
unsigned char ch3;	0 – 255

These declarations each create storage for a character. The standard **char** designation represents the range 0 through 127. An **unsigned char** is a number in the range 0 through 255. A **signed char** covers a range from -127 through 127.

Note: Many compilers allow you to select whether a standard **char** should be treated as signed or unsigned. Check your compiler manual before assuming anything other than the ranges stated above.

A **signed char** is generally used when you need a small integer number. An **unsigned char** variable is used more often to represent an actual character, since characters are never negative.

A **char** variable can be contained in a single byte of memory. This is the size needed to handle all the standard display characters used in most computing environments. For instance, the ASCII character set contains 128 characters, and the IBM PC character set contains 256 characters, both effectively represented in a **char** or an **unsigned char** variable.

Because **char** variables are actually numbers, there are several ways to assign values. For instance, the three assignment statements in Program P4-12 are functionally equivalent.

P4-12

```
/*
   C program to demonstrate assignments to
   char variables.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   char ch;      /* declare the char variable */

   ch = 'A';     /* assign the capital letter A */
   printf("The variable ch is %c\n", ch);
   ch = 65;      /* the ASCII value of capital A */
   printf("The variable ch is %c\n", ch);
   ch = 0x41;    /* the hexadecimal value of 65 */
   printf("The variable ch is %c\n", ch);
}
```

Because char variables are numbers, they can be incremented, decremented, and operated upon just like their integer counterparts. For example, run Program P4-13. (Note that integers have numerous format specifiers, but char variables always use %c, in both **printf( )** and **scanf( )** format strings.)

P4-13

```
/*
   C program to demonstrate math operations on
   char variables.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   char ch1, ch2;  /* declare char variables */
   int i;          /* integer variable */

   printf("Enter a letter: ");
   scanf("%c",&ch1);

   ch2 = ch1+1;  /* increment and assign the character */
```

```
    printf("\n The letter after %c is %c\n",ch1,ch2);

    printf("Now enter a number: ");
    scanf("%d",&i);

    ch2 = ch1 + i;  /* add the number input by the user */
    printf("\n The letter that is %d past %c is %c\n",
           i,ch1,ch2);

}
```

Special-purpose Characters

C uses several special-purpose characters, which can be assigned to any char variable in the same way as ordinary characters. Special-purpose characters have a special meaning when they are displayed or printed.

Character Representation	Use in C Programs
\n	Start a new line (go to beginning of next line)
\t	Insert a horizontal tab
\b	Backspace (move left one character)
\r	Carriage return (go to beginning of current line)
\f	Form feed
\a	Sound an audible alert tone
\0	Insert a NULL character (ASCII 0)
\\	Insert a \ character
\'	Insert the prime character
\"	Insert the quote character
\ddd	Insert the bit pattern with octal value ddd

The use of \n in **printf()** statements you have seen so far in this text is an example of the use of special characters. The \n character is used to move the cursor to the beginning of the next line on the screen. Assigning special characters to a variable is very simple, as shown in Program P4-14.

P4-14

```
/*
   C program to show assignment of special characters.
*/
#include <stdio.h>

main()
{
   char ch;  /* declare the char variable */

   ch = '\n'; /* make the assignment */
   printf("ch is carriage return: %c", ch );
         /* no '\n' is needed in the printf() format
            string, because ch performs the new line function
            for you */
   printf("This is \t tabbed a \t few times! \n");
         /* tab characters ('\t') can be used for spacing
            your output. */
}
```

Here is the output from Program P4-14:

```
C:\CDT>p4-14
ch is carriage return:
This is          tabbed a        few times!

C:\CDT>
```

Characters are most useful when you can group them together into arrays called strings. You'll learn about these in Chapter 6.

FLOATING POINT NUMBERS

Floating point numbers are often called "real" numbers. They are used in many applications where better resolution is required than is available with integers. For instance, any business application requires the tracking of cents, not just dollars.

Declarations

There are two standard floating point variable types in C—the **float** and the **double**. The **double** also has a long version—the **long double**. The following are the standard declarations:

```
float f;
double d;
long double l;
```

The range covered by these variables is presented in three parts: *magnitude, accuracy,* and *precision*. First, a range of magnitude is given—how large (positive) or small (negative) the whole number portion of the number can get. Second, a range of the accuracy of the number is given—how close to 0 can it get. Third, a precision requirement is provided for a particular number of decimal places; this represents the number of places to the right of the decimal point that can be guaranteed accurate.

The **float** variable has a numeric range from -10^{e37} power to $+10^{e37}$, and an accuracy range from -10^{e37} to $+10^{e37}$. The required precision is six decimal places.

The **double** and **long double** variables have the same minimum range and accuracy as a **float**, but provide at least ten decimal places of precision.

As with integer variables, floating point variables have several format specifiers for **printf()** and **scanf()**, based on the variable types and whether exponential (scientific) notation is needed:

Format Specifier	Variable Type
%f	Float or double
%Lf	Long double
%e	Double, in exponential form
%E	Same as %e, but with uppercase E in output, rather than a lowercase e
%Le	Long double, in exponential form
%LE	Long double, in exponential form
%g	Double; use %f or %g based on size of value
%G	Double; same as %g but uses an uppercase E instead of a lowercase e in the output
%Lg	Long double
%LG	Long double

Operations

Floating point numbers have a full range of operations, similar in nature to the integer operations. Try running the next program file now, Program P4-15; its examples demonstrate these operations' capabilities.

P4-15

```
/*
   C program to demonstrate floating point numbers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   /* The next three lines declare 3 "regular"
      float variables, 3 doubles, and 3 long doubles.
      */

   float       f1, f2, f3;
   double      d1, d2, d3;
   long double   ld1, ld2, ld3;

   f1 = 3353.5015;
   f2 = 1318.445;
   f3 = (f1 * f2) / 0.002;
   printf(" f1 = %10.5f, f2 = %10.5f, f3 = %10.5f\n",
      f1, f2, f3);   /* print the values */
   printf(" Different notations: \n");
   printf(" f3 = %10.5f, f3 = %10.5e, f3 = %10.5g\n",
      f3, f3, f3);   /* print the total again */

   d1 = 12000.03303;
   d2 = 322454.12;
   d3 = (d2 / d1) + (3.5 * (d2-d1));
   printf(" d1 = %10.5f, d2 = %10.5f, d3 = %10.5f\n",
      d1, d2, d3);   /* print the values */

   ld1 = .000345;
   ld2 = 8.0002301;
   ld3 = (ld2 * ld1) + (ld2 / ld1);
   printf(" d1 = %10.5Lf, d2 = %10.5Lf, d3 = %10.5Lf\n",
      ld1, ld2, ld3);   /* print the values */
   printf(" Other Notations: \n" );
   printf(" d3 = %10.5Lf, d2 = %10.5LE, d3 = %10.5LG\n",
      ld1, ld2, ld3);   /* print the values */

}
```

The result of Program P4-15 is shown below:

```
C:\CDT>p4-15
 f1 = 3353.50150, f2 = 1318.44500, f3 = 2210703642.58375
 Different notations:
 f3 = 2210703642.58375, f3 = 2.21070e+009, f3 = 2.2107e+009
 d1 = 12000.03303, d2 = 322454.12000, d3 = 1086616.17550
 d1 =    0.00035, d2 =    8.00023, d3 = 23189.07551
 Other Notations:
 d3 =    0.00035, d2 = 8.00023E+000, d3 =      23189

C:\CDT>
```

You may notice a slight difference in the **printf()** statement in this example. The integer specifiers (based on **%d**) have been replaced by floating point specifiers (based on **%f**). The L in **%Lf** denotes a **long** value, and is used with the **long double** examples.

The 10.5 is a size specifier, used with floating point numbers to denote the total width of the space for the number (10 in this case), including the decimal point and the number of digits showing to the right of the decimal point (5 in this case). Thus a value of 23.456544 would be displayed as

```
23.45654
```

The width specifier is a minimum value for the size of the data to print. A value of 12345.67 with a format specifier of %5.2f would print as

```
12345.67
```

even though it is eight characters wide.

Tip: A standard designation for a monetary display might be %12.2f, which gives a width of 12, with 2 digits (the pennies) after the decimal point.

ENUMERATIONS

Enumerations are specialized integer variables in C. Enumerations can be used as a simple method of limiting values for a particular variable;

they also make code easier to read. Program P4-16 defines an integer to keep track of the current seasons of the year:

P4-16

```
/*
   C program to define seasons as integers.
   By: L. John Ribar, C DiskTutor
*/

main()
{
   int season;
   int fall, winter, spring, summer;

   fall = 0;
   winter = 1;
   spring = 2;
   summer = 3;

   season = fall;
   printf("The season is %d\n", season );
}
```

The method shown above is rather bulky, takes a lot of typing, and can become hard to read if you have more options. The enumeration technique is a much more elegant method in C. An enum variable is created in this next example, Program P4-17.

P4-17

```
/*
   C program to demonstrate enum variables.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   enum seasonType { fall, winter, spring, summer };

   enum seasonType season;

   season = fall;
   printf("The season is now %d\n", season );
}
```

After this declaration, the variable **season** is equated to the **fall** enumeration, which is actually a value of 0. Winter, spring, and **summer** have values of 1, 2, and 3, respectively. This code performs the same function as that of Program P4-16, but in a more elegant and readable manner.

Since the enumeration is actually a numeric variable, the increment and decrement operations are also available. The next example, Program P4-18, shows how to move you from one season to the next.

P4-18

```
/*
   C program to move between the seasons.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   enum seasonType { fall, winter, spring, summer };

   enum seasonType season;

   season = fall;
   printf("The season starts as %d\n", season );
   season++;       /* season is now equal to "winter" */
   printf("The season has changed to %d\n", season );
   season++;       /* season is now equal to "spring" */
   printf("The season has changed to %d\n", season );
   season++;       /* season is now equal to "summer" */
   printf("The season has changed to %d\n", season );
   season++;       /* season is now undefined */
   printf("The season has changed to %d\n", season );
}
```

Incrementing an enumeration variable does not, however, wrap around to the first value after the last value is passed. As a result, **summer+1** does not equal **fall**, as you can see in what appears after Program P4-18 is run, shown below:

```
C:\CDT>p4-18
The season starts as 0
The season has changed to 1
The season has changed to 2
The season has changed to 3
```

```
The season has changed to 4

C:\CDT>
```

LOGICAL (BOOLEAN) VARIABLES

Integers may also be used as *logical,* or *boolean,* variables. These are variables used to represent true and false answers. In C, an integer represents false if it has a value of zero. Any other value represents true.

C also supports a group of operations on boolean values, both as variables and as results of other computations or comparisons. These operations include equality, several inequalities, and the AND, OR, and NOT operations. Run this next program, numbered P4-19, and examine its results, using several different values for integer a.

P4-19

```
/*
   C program to demonstrate boolean numbers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int a, b, c;   /* integers used as booleans */

   printf("Enter a number: ");
   scanf("%d",&a);  /* get the first number */
   b = ( a > 0 );
      /* this line sets b to 1 (True) if a is greater
         than 0, otherwise b is set to 0 (False) */

   printf(" a is %d, b is %d \n", a, b);

   c = a || b;   /* if a or b is true, c is also */
   printf(" c is now %d \n",c);

   c = a && b;   /* if a and b is true, so is c */
   printf(" c is now %d \n",c);
   c = ! a;      /* c is set to the opposite of a:
                    if a is true, c is false, else
                    if a is false, c is true. */
   printf(" c is now %d \n",c);

}
```

From Pascal to C

If you know Pascal, this table will help you compare and understand C variable types:

C Variable Type	Pascal Equivalent
int	Integer
unsigned int	Word
long int	LongInt
char	Char
unsigned char	Byte
float	Real

VARIABLE TYPE MANIPULATION

There will be times when the basic variable types provided by C are used in your programs for specific purposes, and you will want to have better documentation of their use. There will also be times when you need to pass data between variables of differing types. For this reason, C provides the **typedef** and **cast** facilities.

RENAMING BASIC TYPES WITH TYPEDEF

The C language contains many types of integers. In some cases, you may want to "rename" the integer type with a more descriptive term. This will serve two purposes: making the program more readable, and allowing variable types to change if necessary for different hardware platforms.

One of the ways to rename the data types is with the **typedef** statement. A common **typedef** is for the boolean type, since there is no special representation in C. Thus, you might use

```
typedef int BOOL;
```

Then within your program you can declare variables of a new type: BOOL. Because of the **typedef** statement, *you* know this is actually an integer, and

your program is now much easier to understand. Others who read it will quickly understand that the BOOL variables are being used to hold true and false values.

> Tip: Although it is not required, it is common C style to express **typedef** names with uppercase letters, as in the BOOL example just above. This helps you to quickly pick out these types of variables during debugging and maintenance of your programs.

The **typedef** statement also aids in portability. For instance, although an **int** and a **short int** are usually the same size with most PC compilers, on some machines they may not be. Using a **typedef** when assigning one of these variable types may save you from problems when running your program on another machine.

Some more interesting uses of **typedef** arise when you redefine pointers and structures, as covered in Chapter 6.

CASTING ONE TYPE TO ANOTHER

Often, data must be passed between different types of variables. A **cast** is used in these cases to override the compiler's default rules for type conversion. The **cast** statement is also used when these rules need to be explicitly stated.

A **cast** is set by putting the desired type name in parentheses in front of the actual variable to translate. For instance, you can assign a small integer to a character value—see Program P4-20.

P4-20

```
/*
   C program to demonstrate simple casting.
   By: L. John Ribar, C DiskTutor
*/

main()
{
   char ch;
   short s;

   s = 65;
   ch = (char) s;
}
```

This tells the compiler that the data currently in the s variable should be formatted to fit the ch variable, as a char, as the assignment is made. To see several more examples and get more familiar with the use of casting, load and compile Program P4-21.

P4-21

```
/*
   C program to demonstrate more casting options.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int      i, j;  /* declare some variables */
   float    k, m;

   /* assign some initial values */

   i = 10000;
   k = 200.55;

   /* now move the values around a bit */

   j = (int) k;       /* j = 200 */
   printf("k is %8.2f, but j was assigned %d\n", k, j );

   j = i / (int) k;
   printf("j is now %d\n", j );

   m = (float) i;   /* m = 10000.0 */

   m = ((float) i) / k;
   printf("m has been assigned a value of %8.2f\n", m);
}
```

Here is the expected output from the above program:

```
C:\CDT>p4-21
k is   200.55, but j was assigned 200
j is now 50
m has been assigned a value of    49.86

C:\CDT>
```

In this example, variables have been assigned values that might otherwise have been unavailable. This was done by casting the values from one data type into another.

Remember: It is safest to **cast** a variable into another variable if the destination has an equal or greater range than the source. For instance, casting from an **int** to a **long int** or **float** will be safe, but if you **cast** from a **double** to a **short int** you may lose some of the resolution of the original number.

SUMMARY

- Integers are variables used to hold counting numbers. You may have long and short integers, which provide lesser or greater ranges for your numbers.
- In C, basic mathematical operations can be performed on numeric variables—both integers and floating point numbers.
- Special operations can be performed on integers, including incrementing and decrementing, shift left and shift right, and bit-level operations.
- Shortcuts are available to shorten assignments to variables when simple operations are being performed.
- Characters are treated like small numbers in C.
- There are special characters in C that perform special functions without much programming effort.
- Floating point variables store real numbers.
- Enumerations help control possible values for a variable.
- The **typedef** and **cast** statements can improve readability in your programs.

MINI-CASE STUDIES

Here are two small case studies to test your knowledge and understanding of the concepts presented thus far. Use these two problems as the basis for your first two real programs. Follow the software design steps you have learned, and write C programs to perform the designated tasks in each problem. Example solutions are given in Chapter 12.

MINI-CASE 1

Develop a program to read a number entered at the keyboard (use a **scanf()** statement), and print out the value in decimal, octal, and hexadecimal numbers. This operation will be useful later in a full case study.

MINI-CASE 2

Develop a program to read a number of minutes entered at the keyboard by the user, and convert it into hours and minutes. Then print the results. You may want to try creating another program to reverse this process; ask for a number of hours and minutes, and convert it into minutes.

C H A P T E R

FLOW CONTROL

Now that you know how to create and manipulate several types of variables, it's important that you learn how to control the *flow* of a program.

Consider, for example, a program that writes payroll checks for 500 employees. Since each check has the same basic format, your program has to include the same sequence of statements 500 times! To control the order in which program statements are executed, you use *flow control statements.* By including in your payroll program a flow control statement that allows *looping,* the group of check processing statements in the program need only be written once, and then executed for each of the 500 employees.

In previous chapters you've been introduced to the concept of a block of code; in this chapter you'll learn more about blocks, and examine the function of the two main types of flow control statements. One type is used to make a decision and then perform selected code based on the results. The second type is used to repeat sections of code multiple times, in a controlled manner. Other control statements are also introduced here.

UNDERSTANDING PROGRAM BLOCKS

In most programming languages, including C, a group of statements executed together is known as a *block*. In C, a block is surrounded by the curly braces { and }. Blocks are an important component of program flow.

Once a flow control statement makes a decision in a program, either a single line or an entire block of code will be executed, as defined by the statements between the braces. For instance, in Program P5-1, one of two printf( ) statements will be executed, depending on the outcome of the if statement (you'll learn about if statements in the coming section about decision making).

P5-1

```
/*
   C Program to demonstrate if statements.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int a;    /* define the variable */

   printf("Enter an integer value: ");
   scanf(" %d", &a);    /* read value from the user */

   if (a==1)    /* check the value of a */
      printf("a is one\n");
   else
      printf("a is not one\n");
}
```

In the next example, Program P5-2, the series of commands within the curly braces are executed together as a block. Notice that this block contains other blocks.

P5-2

```
/*
   C Program to demonstrate blocks.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
```

```
{
   int c, d;              /* declare some variables */

   printf("Please enter 1 or 2: ");
   scanf(" %d", &c );   /* get a value from the user */

   if (c==2)              /* check the value of c */
   {
      d = 2;
      printf("Everything is now 2\n");
   }
   else
   {
      d = 3;
      printf("Nothing is 2 now! c is %d and d is %d\n",
            c, d);
   }
}
```

Tip: Notice that the block of code between the main pair of braces is indented. This is not required, but it is a good practice. Indenting makes the code more readable, and it better delineates the block as a cohesive unit. You may want to review Chapter 2's discussion on indenting and program style.

USING DECISION-MAKING STATEMENTS

Decision-making control statements are used when a decision is necessary before the program can continue. An if statement checks the value of a *condition*. When the condition is true, specific code is executed; when the condition is false, no code is executed. If there is an else clause, and the original condition is false, the code following the else statement is executed.

A condition can be a variable with a true or false value. Or, a condition can be a comparison between two values or variables, or the return value from a function. When using comparisons as a condition, the following types of comparisons can be made:

Comparison	Meaning
a==b	Is a equal to b?
a<b	Is a less than b?
a<=b	Is a less than or equal to b?
a>b	Is a greater than b?
a>=b	Is a greater than or equal to b?
a!=b	Is a not equal to b?

These comparisons return true or false, based on the results of the actual comparison.

A switch statement can check the value of a variable, and then jump (go to) a specific set of code based on that value.

THE IF AND ELSE STATEMENTS

Use an if statement when a decision is to be made that will determine whether certain lines of code will be executed. The if statement has the following general forms:

```
/* No 'else' clause; single statement to execute */
if (condition)
   statement1;

/* No 'else' clause; block of statements to execute */
if (condition)
{
   statement1;
   statement2;
   /* could be more statements here, of course */
}

/* Has 'else' clause; block of statements to execute
   under if and else */
if (condition)
{
   statement1;
   statement2;
}
else
{
```

```
   other_statement1;
   other_statement2;
}
```

Note: Notice that the foregoing examples all have two statements within the block; C actually allows any number of statements within a block, including none.

An if statement is composed of a *condition* and one or more *statements.* The *condition* is always surrounded by parentheses, as shown in the above general form description. If only one *statement* needs to be executed as a result of the *condition,* then you can omit the curly braces. Otherwise, use a standard C block, including the curly braces. For readability, indent the *statement* or block from the if and else statements.

The else portion of the statement is optional. Use it when you want an alternate set of code to be executed if the original condition is not true.

The *condition* portion of the if statement is any statement or equation that returns a value. This can include a test for equality or inequality; the name of an integer variable with a value of zero (false) or a value not equal to zero (true); or a call to any other type of statement that returns a zero or nonzero value. These options exist because of how C handles boolean values. Any equation with a value of zero is false; any other value is considered true. The following program, Program P5-3, shows some of these options in action:

P5-3

```
/*
   C program to show if conditions.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

typedef int BOOLEAN;    /* rename for clarity */

main()
{
   BOOLEAN more;
   int a, b;

   more = 0;
   a = 0;

   printf("Enter a value for a: ");
```

```
   scanf(" %d", &a);
   if (a == 0)
      printf("a is zero \n");
   else
   {
      printf("a is ");
      b = a % 2;  /* find out if it is odd or even */
      if (b)
         printf("odd\n");
      else
         printf("even\n");
   }

   more = ( a > 10 );

   if (more)
      printf("You wanted more!\n");
   else
      printf("You have had enough.\n");
}
```

When a condition yields a true value, the very next statement or block is executed. If there is an else option, it is executed if the original condition was false. If there is no else option, and the original option was false, nothing will be executed.

Now, look at this next program, numbered P5-4. You will see a simple if statement that will be used in the following discussion.

P5-4

```
/*
   C program to demonstrate if statements.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int month;    /* declare the integer variable */

   printf("Enter a number: ");
   scanf("%d",&month);

   if (month == 1)
      printf("This is January\n");
}
```

The foregoing simple program shows an if statement with a single result. If the variable **month** has a value of 1, then the **printf( )** statement will be executed. But what happens if **month** is 2? In this example, nothing would happen. A slight change, however—adding an else statement—will give us some more information, as shown in Program P5-5:

P5-5

```
/*
   C program to print whether the given month is January.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int month;    /* declare the variable */

   printf("Enter a number: ");
   scanf(" %d",&month);    /* read a month from the user */

   if (month == 1)
      printf("This is January\n");
   else
      printf("This is not January!\n");
}
```

This use of else can be extended even further; an else statement can contain another if statement. Look at Program P5-6 to see how if-else statements can be strung together.

P5-6

```
/*
   C program with lots of if's and else's
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int month;    /* declare the variable */

   printf("Enter the number of a month: ");
   scanf(" %d",&month);
```

```
    /* now check for the correct month */
    if (month == 1)
       printf("This is January\n");
    else if (month==2)
       printf("This is February\n");
    else if (month==3)
       printf("This is March\n");
    else if (month==4)
       printf("This is April\n");
    else if (month==5)
       printf("This is May\n");
    else if (month==6)
       printf("This is June\n");
    else if (month==7)
       printf("This is July\n");
    else if (month==8)
       printf("This is August\n");
    else if (month==9)
       printf("This is September\n");
    else if (month==10)
       printf("This is October\n");
    else if (month==11)
       printf("This is November\n");
    else if (month > 12)
       printf("This is NOT a month! You should use 1 thru
              12\n");
    else
       printf("This must be December!\n");

}
```

Try running the foregoing program with different values for the **month** variable, and observe how the different printf( ) statements are executed.

Caution: The condition part of an if statement gives many beginning C programmers the most problems. Keep in mind that == is used to compare values for equality, and = is for assignment. In C, (a=10) is a valid condition statement; it assigns a the value of 10, and returns that value (10). The condition is therefore true, because the value is nonzero. And (a==10) is a valid condition statement, too. If a is equal to 10, a true (1) is returned; otherwise, false (zero) is returned. Be careful that you use the right operation in your if statements.

From Pascal to C

The if and switch statements in C are much like the IF and CASE statements in Pascal. Compare Programs P5-7 and P5-8:

P5-7

```
/*
   C program to show if and switch statements.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>    /* allow for screen input/output */

main()
{
   int month;            /* declare an integer variable */
   printf("Enter the month: ");      /* ask for input */
   scanf(" %d", &month );            /* get the input */
   if (month < 1)          /* check for invalid inputs */
      printf("That month is too small!\n");
   else if (month > 12)
   {
      printf("That month is too big.\n");
      month = 12;        /* put user in holiday spirit! */
   }

   switch (month)              /* check for actual value */
   {
      case 1:
      case 2:
         printf("Boy, is it cold out!\n");
         break;
      case 12:
         printf("Happy Holidays!\n");
         break;
      case 3:
      case 4:
      case 5:
         printf("Time to plant your garden.\n");
         break;
      case 6:
      case 7:
      case 8:
         printf("Let's go out to the ball game.\n");
         break;
      case 9:
      case 10:
      case 11:
```

From Pascal to C (*continued*)

```
         printf("Time to harvest the vegetables.\n");
          break;
       default:
          printf("I do not know what to say!\n");
          break;
   };                      /* end of switch statement */
}
```

P5-8

```
(*
   Pascal program to show IF and CASE statements.
   By: L. John Ribar, C DiskTutor
*)
program CaseDemo;

uses CRT;              (* allow for screen input/output  *)

VAR
   month : INTEGER;      (* declare an integer variable  *)
BEGIN
   Write('Enter the month: ');          (* ask for input *)
   ReadLn( month );                     (* get the input *)
   IF month < 1 THEN         (* check for invalid inputs *)
      WriteLn('That month is too small!')
   ELSE IF month > 12 THEN
   BEGIN
      WriteLn('That month is too big.\n');
      month := 12;         (* put user in holiday spirit! *)
   END;

   CASE month OF               (* check for actual value *)
      1, 2:
         WriteLn('Boy, is it cold out!');
      12:
         WriteLn('Happy Holidays!');
      3..5:
         WriteLn('Time to plant your garden.');
      6..8:
         WriteLn('Let's go out to the ball game.');
      9..11:
         WriteLn('Time to harvest the vegetables.');
      ELSE
         WriteLn('I do not know what to say!');
   END;                        (* end of CASE statement *)
END.
```

THE SWITCH AND CASE STATEMENTS

Unfortunately, multiple combinations of if and else statements can become very complicated, and difficult to read. You can alleviate this by using a switch statement.

The switch statement is similar to an if statement with lots of else clauses; however, the result is a more readable piece of C code. For instance, the next program performs the same function as Program P5-6, but Program P5-9 uses a switch statement.

P5-9

```
/*
   C program to demonstrate switch statements.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int month;  /* declare the month variable */

   printf("Enter the number of the month: ");
   scanf(" %d", &month);

   switch (month)   /* check for the matching case */
   {
      case 1  : printf("This is January\n");
                break;
      case 2  : printf("This is February\n");
                break;
      case 3  : printf("This is March\n");
                break;
      case 4  : printf("This is April\n");
                break;
      case 5  : printf("This is May\n");
                break;
      case 6  : printf("This is June\n");
                break;
      case 7  : printf("This is July\n");
                break;
      case 8  : printf("This is August\n");
                break;
      case 9  : printf("This is September\n");
                break;
      case 10 : printf("This is October\n");
```

```
                break;
        case 11 : printf("This is November\n");
                break;
        case 12 : printf("This is December\n");
                break;
        default : printf("This is NOT a month!\n");
                break;
    }
}
```

The **switch** statement does not use the same sort of condition as the **if** statement. In the foregoing example, the **month** variable is being used as the **switch** clause. Because **month** can hold any of several values, several **case** statements are used to select one of the possible values. Another special **case** statement—**default**—is used to cover all unknown values.

When a **switch** statement is executed, the **switch** value is checked against the possible **case** statements. If one matches, the code following that **case** statement is executed. If no match is found, the **default** clause will be used.

The **switch** statement is preferred over multiple **if-else** statements. In the previous example, the result of each **case** statement was only a **printf()** statement, but the processing required under each case statement could easily extend into many other statements. Clearly, organizing all the choices under a **switch** statement makes the program easier to read. Having to thread through many **if** and **else** statements, trying to determine how all the statements are connected, can be very confusing.

The **switch** statement has another peculiarity that can be a help or a hindrance. When a matching **case** statement is found after a **switch** statement, the program jumps to that **case** statement and continues executing code. This means *all* the subsequent **case** statements will also be executed—unless you include a **break** statement to stop the process. You add the **break** statement to the **case** statement so the program can exit the **switch** statement, jumping over other pieces of code that you do not want executed. Without the **break**, all the code following the selected **case** statement will be executed.

Sometimes, however, this can be helpful. Look at Program P5-10:

P5-10

```
/*
   C program to show switch statements that fall through.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int num, add;  /* declare necessary variables */

   num = 5;       /* start with a value of 5 */
   add = 5;       /* we want to increment by 5 */

   switch (add)
   {
      case 5: num += 1;
      case 4: num += 1;
      case 3: num += 1;
      case 2: num += 1;
      case 1: num += 1;
      default : printf("done adding\n");
   }

   printf("num now has a value of %d \n",num);

}
```

This simple example shows the different ways a **switch** statement will execute without a **break** statement. Try running this program; here are the results, showing the output after falling through a set of cases:

```
C:\CDT>p5-10
done adding
num now has a value of 10

C:\CDT>
```

Now add **break** statements, as shown here in Program P5-11, and run the program again.

P5-11

```
/*
   C program to show switch statement with breaks.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int num, add;  /* declare necessary variables */

   num = 5;       /* start with a value of 5 */
   add = 5;       /* we want to increment by 5 */

   switch (add)
   {
      case 5: num += 1;
              break;
      case 4: num += 1;
              break;
      case 3: num += 1;
              break;
      case 2: num += 1;
              break;
      case 1: num += 1;
              break;
      default : printf(“done adding\n”);
   }

   printf(“num now has a value of %d \n”,num);

}
```

Notice how the output changes after breaks have been installed:

```
C:\CDT>p5-11
num now has a value of 6

C:\CDT>
```

All the foregoing examples have handled only integer values in the switch condition. However, because character, enumeration, and boolean values are actually extensions of the integer type, they are just as valid as switch statement conditions. For instance, Program P5-12 demonstrates a simple method for checking the response of a user.

PS-12

```
/*
   C program with character-based switch statement.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   char ch;    /* declare the variable */

   printf("Enter a Y (yes) or an N (no) :");
   scanf(" %c",&ch);  /* read the character from the user */

   switch(ch)    /* what character was entered? */
   {
      case 'y' :
      case 'Y' : printf("Thank you for saying Yes\n");
                 break;
      case 'n' :
      case 'N' : printf("So sorry you said No\n");
                 break;
      default  : printf("I see that you are undecided!\n");
                 printf("You did not choose Yes or No!\n");
                 break;   /* this break is not necessary after
                             default, but is often included as
                             good programming style */
   }

}
```

The foregoing program also demonstrates another feature of the switch statement. In this example, the upper- and lowercase values should produce the same results. By using two case statements—one each for uppercase and lowercase characters—both possible inputs are handled.

Tip: When you're deciding whether to use a group of if statements or a switch statement, consider the nature of the values you are using. An if statement can take any equation that returns a zero or nonzero value, including floating point values. A switch statement, on the other hand, is limited to int, char, or other variable types that have exact (nonfractional) values.

PERFORMING REPETITIVE TASKS

Very often, you will want to write programs that perform functions more than once before they exit. For instance, in the calculator project (Case Study 1 in Chapter 8), you will want the program to perform more than one calculation before exiting. If you don't allow a way for this to happen, the program will have to be rerun for each addition, subtraction, multiplication, and division calculation you wish to perform.

There are also situations where a section of code needs to be reexecuted several times. In some cases, you'll know how many times you want the code to execute before the program is run, whereas in the calculator you'll want to keep going until the user wants to quit.

You can simplify these repetitive tasks by using various C statements, including **while**, **do-while**, and **for** loops.

USING WHILE AND DO-WHILE LOOPS

The **while** and **do-while** loops are very much like **if** statements that can be executed more than once. These two loops have a control statement that determines whether the loop will continue to execute. In an **if** statement, the control statement is only applied the first time through. With an **if** statement, the control statement determines whether the line or block of code will be executed *at all*. With **while** and **do-while** loops, the control statement determines whether the line or block of code will be executed *at another time*.

Nearly identical in purpose, the only difference between a **while** and **do-while** loop is the timing of when the control statement is executed. The general form of these two statements is as follows:

```
/* do statement */

do
   statement_or_block
while (condition);


/* while statement */

while (condition)
   statement_or_block;
```

In a do statement, the body of the loop (the *statement_or_block*) will always be executed *at least once*. The control statement (the *condition*) is checked after each time through the loop. Look at Program P5-13 for an example of a do loop; then compare it to Program P5-14, where a while loop is used.

P5-13

```
/*
   C program to demonstrate the do statement.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i;    /* declare a variable */

   i = 0;    /* initialize the variable */
   do
   {
      printf("i is now %d\n", i);
      i++;
   } while (i<10);
   printf("i is %d at the end.\n", i);
}
```

The execution of Program P5-13 produces the following output:

```
C:\CDT>p5-13
i is now 0
i is now 1
i is now 2
i is now 3
i is now 4
i is now 5
i is now 6
i is now 7
i is now 8
i is now 9
i is 10 at the end.

C:\CDT>
```

In a while loop, the control statement is checked before *each* execution of the loop, so the body of the loop may not be executed at all.

P5-14

```
/*
   C program to demonstrate the while statement.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i;    /* declare a variable */

   i = 0;    /* initialize the variable */
   while (i<10)
   {
      printf("i is now %d\n", i);
      i++;
   };
   printf("i is %d at the end.\n", i);
}
```

Here is the execution of the **while** loop in Program P5-14:

```
C:\CDT>p5-14
i is now 0
i is now 1
i is now 2
i is now 3
i is now 4
i is now 5
i is now 6
i is now 7
i is now 8
i is now 9
i is 10 at the end.

C:\CDT>
```

You can also use either a **do** or **while** statement to provide a simple "forever" loop. If you want a loop to execute continuously (forever), use **(1)** as the control statement, since 1 is always the same as **true** in C. The only way out of the loop in this case is with the **break** statement. (As you learn C, you will notice that the **break** statement is used to exit from several types of loops and control statements.) This **break** exit is always to the

statement outside the current loop. For instance, if you had a loop inside another loop, the break would only exit from the internal loop. Look at Program P5-15 to see how this works.

P5-15

```
/*
   C program with internal and external loops.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int  go;    /* boolean variable */
   int  total, one, two;  /* total and two numbers */
   char ch;    /* the command from the user */

   go = 1;  /* true */

   /* OUTER LOOP */
   while (go)     /* keep going while this is true */
   {
      printf("Enter first number: ");
      scanf(" %d",&one);
      printf("Enter second number: ");
      scanf(" %d",&two);
      printf("Now enter commands to use on these numbers.\n");
      printf("You can use +, -, *, /, or Q to quit, or N \n");
      printf("to enter two different numbers.\n");

      do  /* INNER LOOP*/
      {
         printf(">> ");    /* prompt the user for a command */
         scanf(" %c",&ch); /* this call reads one character
                              from the user, but requires the
                              carriage return to be pressed. */
         switch (ch)
         {
            case '*' : printf(" %d * %d = %d \n", one, two,
                             one*two);
                       break;  /* out of the switch */
            case '/' : if (two == 0)
                          printf("Cannot divide by zero!\n");
                       else
```

```
                    printf(" %d / %d = %d \n", one, two,
                           one/two);
                  break;  /* out of the switch */
          case '+' : printf(" %d + %d = %d \n", one, two,
                           one+two);
                  break;  /* out of the switch */
          case '-' : printf(" %d - %d = %d \n", one, two,
                           one-two);
                  break;  /* out of the switch */
          case 'Q' : go = 0;  /* ready to quit now! */
                  break;  /* out of the switch */
        }

        if ((ch == 'Q') || (ch == 'N'))
          break;  /* leave the do loop if user wants to quit
                     or is in need of new numbers */

      } while (1);  /* end of the do loop */

    }  /* end of the while loop */

    printf("finished...\n");
}
```

The following is some output from the simple calculator program shown in Program P5-15:

```
C:\CDT>p5-15
Enter first number: 65
Enter second number: 13
Now enter commands to use on these numbers.
You can use +, -, *, /, or Q to quit, or N to
enter two different numbers.
>> +
 65 + 13 = 78
>> -
 65 - 13 = 52
>> /
 65 / 13 = 5
>> N
Enter first number: 22
Enter second number: 33
Now enter commands to use on these numbers.
You can use +, -, *, / or Q to quit, or N to
enter two different numbers.
```

```
>> +
 22 + 11 = 33
>> q
Finished. . .

C:\CDT>
```

You can see how the if, while, do-while, and switch statements work together with the break statement to provide several methods of control over how the program will execute. Run several different iterations of Program P5-15 and watch how the loops execute within one another.

USING FOR LOOPS

Use a for loop to execute a statement or block multiple times. To use a for loop, you use an index variable, which keeps track of how many times you've been through the loop. You will then initialize the variable (give it an initial, or starting, value), set a limit that will cause the loop to exit, and define how the variable will be changed each time through the loop.

Look at the simple example of a for loop in Program P5-16:

P5-16

```
/*
   C program to print out 10 values.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i;   /* declare the variable */

   for (i=0; i<10; i++ )   /* ten times */
      printf("Value of i is %d\n",i);
}
```

In the above example, the i variable (the index) is initialized to 0 (i=0). Each time through the loop, a test is made to be sure that i is still less than 10 (i<10). If i is less than 10, the loop is executed, and then i is incremented using i++.

Run the program now to see what happens. Your output will look like the following:

```
C:\CDT\CHAP5>p5-16
Value of i is 0
Value of i is 1
Value of i is 2
Value of i is 3
Value of i is 4
Value of i is 5
Value of i is 6
Value of i is 7
Value of i is 8
Value of i is 9

C:\CDT\CHAP5>
```

Note: In the output from Program P5-16, you may notice that the numbers 0 through 9 are printed, but not 10. This is because the **i<10** condition was met. Once **i** was greater than or equal to 10, the **for** loop was exited.

The **for** condition consists of three parts, separated by semicolons. The three parts are all optional—in fact, the simplest form of the **for loop** is a forever loop:

```
for (;;)
{
   statement;
}
```

This loop will continue to execute the statement forever, unless a **break** statement is encountered. (This is another use for the **break** statement.) To understand why this loop will continue forever, you must understand the three parts of the *condition*:

```
for ( init ; control ; increment )
   statement_or_block;
```

The *init* statement is executed before the loop starts. This is usually done to initialize the index variable, or any other variables needed within the loop. When no statement is entered here, nothing will be initialized.

The *control* statement is executed before each iteration of the loop. This is generally done to limit the number of times the loop will execute. If the value of the *control* statement is nonzero (true), then the loop continues to

execute. When no statement is given here, a true value is assumed, so the loop will execute again.

The *increment* statement is executed at the end of each journey through the loop. This is usually done to increment the index variable. When no statement is entered here, nothing will be executed, and the next test of the control statement will occur. The most common use of the *increment* statement is to increment the index variable, using i++ or the equivalent. You can, however, perform other operations here. To multiply the index variable by 2 each time through the loop, for instance, your **for** statement might look like this:

```
for (i=0; i<100; i=i*2 )
   statement_or_block;
```

There are two special statements you can use within a **for** loop to modify its execution. The **break** statement causes the program to jump out of the **for** loop. The **continue** statement skips the rest of the block that makes up the loop, and executes the next increment statement for the loop. Try the next program to see how the **continue** and **break** statements affect a **for** loop.

P5-17

```
/*
   C program to demonstrate break and continue.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i;    /* declare the variable */

   for (i=0; i<20; i++)
   {
      if (i==4)
         continue;      /* don't print this one */
      if (i==12)
         break;         /* this will jump out of the loop */
      printf("loop is now on %d\n",i);
   }
   printf("That is all. i is now = %d \n",i);
}
```

The output for Program P5-17 is shown here:

```
C:\CDT\CHAP5>p5-17
loop is now on 0
loop is now on 1
loop is now on 2
loop is now on 3
loop is now on 5
loop is now on 6
loop is now on 7
loop is now on 8
loop is now on 9
loop is now on 10
loop is now on 11
That is all. i is now = 12

C:\CDT\CHAP5>
```

Note that a for loop is actually a specialized version of a **while loop**. The next two loops are functionally identical:

```
/* the for loop version */

for (i=0; i<10; i++)
   printf("i = %d \n",i);


/* the while version */

i = 0;                    /* initializer */
while (i<10)        /* control statement */
{
   printf("i = %d \n",i);
   i++;                   /* incrementor */
}
```

Notice the same three required statements—an initializer, a control statement, and an incrementor. This is, however, a specialized use of a **while** statement; while statements can be used in many other ways.

OTHER FLOW CONTROL STATEMENTS

The most commonly used flow control statements in C are the decision and repetition control statements you have learned about so far in this chapter. Here are two others that you will want to know: **goto** and **return**.

USING GOTO STATEMENTS

A **goto** statement provides a quick means of jumping to another point within a program (a *label*). Although **goto** statements provide great flexibility, their use is generally frowned upon because it can be difficult to trace their execution. In contrast, the paired curly braces around an if block are easy to locate. Finding the destination label needed by a **goto** statement, however, can be an arduous task.

Tip: The best time to use a **goto** statement is never. These statements can be terrible to follow during debugging, so use them only as a last resort.

Nevertheless, there are times when the **goto** statement can be useful. Look at the next program. This is the same as Program P5-15, except that a **goto** is used to quickly exit the program (it is in the "Q" case statement). Notice that the format of the label needed by the **goto** is simply the label name followed by a colon.

P5-18

```
/*
   C program demonstrating goto within multiple loops.
   By: L. John Ribar, C DiskTutor
/*
 #include <stdio.h>

main()
{
   int  go;   /* boolean variable */
   int  total, one, two;  /* total and two numbers */
   char ch;   /* command entered by the user */
```

```
go = 1;  /* true */

/* OUTER LOOP */
while (go)    /* keep going while this is true */
{
   printf("Enter first number: ");
   scanf(" %d",&one);
   printf("Enter second number: ");
   scanf(" %d",&two);
   printf("Enter commands to use on the numbers.\n");
   printf("Use +, -, *, /, or Q to quit, or N to \n");
   printf("enter two different numbers.\n\n");

   do
   {
      printf(">> ");    /* prompt user for a command */
      scanf(" %c",&ch); /* reads one character from
                           the user */
      switch (ch)
      {
         case '*' : printf(" %d * %d = %d \n", one, two,
                           one*two);
                    break;  /* out of the switch */
         case '/' : if (two == 0)
                       printf("No divide by zero!\n");
                    else
                       printf(" %d / %d = %d \n", one, two,
                           one/two);
                    break;  /* out of the switch */
         case '+' : printf(" %d + %d = %d \n", one, two,
                           one+two);
                    break;  /* out of the switch */
         case '-' : printf(" %d - %d = %d \n", one, two,
                           one-two);
                    break;  /* out of the switch */
         case 'Q' : goto done;  /* quick exit! */

   }
```

```
         if (ch == 'N')
            break;  /* leave the do loop if user wants
                       new numbers */

      } while (1);  /* end of the do loop */

   }  /* end of the while loop */

done:    /* here is the label */

   printf("finished...\n");
}
```

USING RETURN STATEMENTS

The **return** statement is used to return control of a program from a function to the main program. You will learn more about **return** in Chapter 7.

SUMMARY

This chapter introduced you to the most common and useful methods of controlling the flow of a C program. Through the use of the following statements, you have great control over the execution of your C programs:

- **if-else**
- **switch**
- **for**
- **while**
- **do-while**
- **goto**

MINI-CASE STUDIES

Here are two small case studies to test your knowledge and understanding of the concepts presented thus far. Your solutions to these problems will form a good basis for the full Case Studies, which start in Chapter 8. Follow the usual design steps as you work on these problems, and write C programs to perform the necessary tasks. Example answers are given in Chapter 12.

MINI-CASE 3

There is a function in the standard C library that allows you to get the amount of time that has elapsed since your program started (in timer ticks) from the system. The function is called **clock( )**. It returns a variable of type **clock_t**. To determine the number of seconds since your program started, divide the **clock_t value by** CLOCKS_PER_SEC (a value included in the time.h header file). A call to **clock( )** looks like this:

```
timerTicks = clock( );
```

Remember: To use the **clock( )** function, be sure to **#include** the time.h header file mentioned above.

There is another function of use for this program. The function **getchar( )** reads a single character from the user. Using **clock( )** and **getchar( )**, write a program which waits for the user to press a key and then prints the amount of time your program has been running, in seconds. This could be used as a simple stopwatch!

MINI-CASE 4

In another version of the problem in Mini-Case 3, try printing the elapsed time in words, as in *four minutes and five seconds*, instead of *245 seconds*.

C H A P T E R

ADVANCED DATA MANIPULATION

This chapter introduces some of the more difficult, and more useful, data storage and manipulation features of the C language. The concepts explained here—arrays, pointers, structures, and unions—are handled by C in a way that is somewhat less straightforward than in some other languages. These structures may take a little longer for you to grasp, but be sure to persist; understanding them will make you a better C programmer, and add greater power and depth to your programming repertory.

UNDERSTANDING ARRAYS

An array is one of the most commonly used data structures in any programming language. Simply defined, an *array* is a group of objects with similar structure but different values. Let's examine this more closely.

An array is made up of one or more objects. Each object has the same structure as the others, but also has its own value. For instance,

an array of integers contains several integers; all the integers have the same structure, and each has its own value.

In this first example, Program P6-1, five numbers are read from the user, calculations are performed, and results are printed out. Each of the five numbers is stored in a separate **int** variable. Type or load this program into DTE and run it; notice how much work is involved for just five variables.

P6-1

```
/*
   C program to work with 5 integers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

          Five separate declarations
main()
{
   int i1, i2, i3, i4, i5;  /* Declare the integers */

   /* Read in the numbers */
   printf("Enter number 1: ");
   scanf(" %d",&i1);
   printf("Enter number 2: ");
   scanf(" %d",&i2);
   printf("Enter number 3: ");           Five prompts and inputs
   scanf(" %d",&i3);
   printf("Enter number 4: ");
   scanf(" %d",&i4);
   printf("Enter number 5: ");
   scanf(" %d",&i5);

   /* Now perform the calculations */
   i1 = i1*i1 + i1;
   i2 = i2*i2 + i2;
   i3 = i3*i3 + i3;     Five calculations
   i4 = i4*i4 + i4;
   i5 = i5*i5 + i5;

   /* and finally, print the results */
   printf("The result for number 1 was %d\n",i1);
   printf("The result for number 2 was %d\n",i2);
   printf("The result for number 3 was %d\n",i3);   Five outputs
   printf("The result for number 4 was %d\n",i4);
   printf("The result for number 5 was %d\n",i5);

}
```

Now try the next program, P6-2, and notice how much simpler it is to use five numbers in an array. Though you haven't yet learned the specifics of using arrays, you can see already in this example how much easier it is to use arrays instead of separately named variables.

P6-2

```
/*
   C Program to demonstrate a simple array of integers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>
                          One declaration
main()
{
   int a[5];    /* Declare an array of 5 integers */
   int i;       /* Declare an indexing variable */

   /* Read in the numbers */
   for (i=0;i<5;i++)
   {                                                           One
      printf("Enter number %d: ",i+1);                         prompt
                   /* You need to prompt for number 'i+1'      and
                      so that the user will be asked for       input
                      1 through 5, rather than 0 through 4.
                      This matches the operation of the
                      previous program.
                   */
      scanf(" %d",&a[i]);
   }

   /* Perform the calculations */      One calculation
   for (i=0;i<5;i++)
   {
      a[i] = a[i]*a[i] + a[i];
   }                                   One output

   /* Display the results */
   for (i=0;i<5;i++)
   {
      printf("The result for number %d was %d\n", i+1, a[i]);
   }

}
```

The size of the code (the number of lines you had to type) in this second example is slightly smaller, even with only five variables. Consider what the proportional savings would be if you needed to use 100, or 100,000, integer variables!

SIMPLE ARRAYS

An array is declared by placing a pair of square brackets, [and], after a variable name, and entering within the brackets a number representing the number of elements in the array. Thus, for an array of five integers, you would use the following declaration:

```
int array[5];
```

This creates five integers with these names:

```
array[0]
array[1]
array[2]
array[3]
array[4]
```

Notice that the numbering of the array elements—array[0] through array[4]—starts from zero, not one. This is because C *numbers everything from zero.* Also, it is very convenient in **for** loops, because you know that each element of the array has an index value less than the maximum defined for the array.

Caution: The numbering of array elements is another area where beginning C programmers often have difficulty. C allows you to assign values to any array element, even if it does not exist. For instance, if you declare an array to have ten elements, they are numbered from 0 to 9. However, C will still allow you to assign a value to element 10, which is actually the eleventh element! This will cause unpredictable results in your program. *So remember: Arrays are always numbered from zero.*

You can create arrays for any type of variable. This includes integers, floating point numbers, and characters. It also includes pointers and structures, which will be covered later in this chapter.

The square brackets [and] are also needed for selecting an *array element*. To choose an element, just put its index between the square brackets. If you have an array declared like this:

```
float fa[10];
```

you can select the third element by using

```
fa[2];
```

Again, remember that C always numbers from zero, so the first element in the array **fa[ ]** is **fa[0]**, the second element is **fa[1]**, and the third element is **fa[2]**.

Here are some examples of valid array declarations:

```
int a[10];          /* An array of 10 integers */
char name[20];      /* 20 characters */
float costs[100];   /* 100 floating point numbers */
```

The following declarations are invalid:

```
int a[0];           /* Must be at least 1 element */
char name["size"];  /* Must use integer values */
float other[-10];   /* Negatives are not allowed */
```

When you declare an array, you can immediately give values to the *members* of the array. Place an equal sign after the declaration, followed by the initial values separated by commas and enclosed in curly braces. For instance, the following declaration defines an array containing the number of days in each month:

```
int days[12] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

A special feature of the C language is that when the values of an array are defined immediately, as just shown, then the maximum array size need not be specified. For instance, this declaration performs the same function as the previous one:

```
int days[] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

When the array size is not stated, the maximum value is computed by the compiler when it compiles the code.

Array elements can be used in C statements just as any other variables are used. To assign a value to a member of a floating point array, use a statement like this:

```
fa[2] = 12.43;
```

To use array members in comparison statements, use a statement like this:

```
if (fa[2] < 50.0)
   printf("The number is less than 50.\n");
```

STRINGS

A *string* is a special array in C. A *string array* consists of one or more **char** variables, and the last character always has a value of 0. This is so that any routines written to work with strings know where the string ends. The zero value at the end of the string is often referred to as the *null terminator*.

As with other arrays, the maximum limit of a string array does not need to be specified if the array is initialized during the declaration. The maximum length will be one character greater than the length of the string, to allow for the null terminator. So, in this example,

```
char name[] = "Joe Smith Hardware";
```

the **name** array actually has a declaration of

```
char name[19];
```

representing the eighteen characters in the initial value, plus the null terminator.

Remember: When declaring strings (character arrays) that are not initialized, always add one to the total value, to leave space for the final zero (null terminator).

In the previous example, notice also that the string initialization did not use curly braces. This is because only one value was given. In general, any variable can be initialized when it is declared. Curly braces are needed if more than one value is given. This example could also be written in a

way that requires the curly braces. Since the variable name is actually an array of characters, the declaration could be written as

```
char name[] = {'J', 'o', 'e', ' ', 'S', 'm', 'i', 't', 'h',
     ' ', 'H', 'a', 'r', 'd', 'w', 'a', 'r', 'e', 0 };
```

MULTIDIMENSIONAL ARRAYS

Just as a simple array is a group of objects, a *multidimensional array* is an array of arrays. For instance, let's look at the arrays in the following, Program P6-3. In this example, the name of each month is stored in a string (character array). However, all 12 months are then stored in an array of strings.

P6-3

```
/*
  C program to work with character arrays.
  By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

/*** Declare some global variables. Since they are outside of
     the main() routine, any function can use them. ***/

char MonthName[12][10] = /* 12 months, 10 characters each */
   {
      "January", "February", "March", "April", "May",
      "June", "July", "August", "September", "October",
      "November","December"
   };

int days[] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

main()
{
   int which = 0;  /* Initialize a simple variable */

   printf("Enter the number of the month: ");
   scanf(" %d",&which);
   while (which)   /* If which = 0, this will be false */
   {
      printf("\n Month %d is called %s, and has %d days.\n",
              which, MonthName[which-1], days[which-1]);
      printf("\n Enter the number of the month: ");
```

```
        scanf(" %d",&which);
    }
}
```

Keep in mind that C does not check the array index you use. Thus, in Program P6-3, you can enter a month 13, for example, or 100, and C will not give you an error. The results you get, however, will be unpredictable. Try running Program P6-3 again, and enter some values other than 1 through 12, to see what happens.

Note: Because arrays are always numbered from zero, when the user enters a number from the keyboard, the program must subtract 1 from it to find the correct item in the array.

The initialization of the strings in Program P6-3 is somewhat different, in that a size has been used that does not match the lengths of the strings. Since the lengths of the month names are not all the same, a maximum value of 10 was used when declaring the arrays. In cases where the month name was less than nine characters, the extra characters are unused but still available.

A more efficient way to declare these strings would have been as an array of 12 *pointers*, each pointing to a string of the correct length. Pointers are discussed next.

POINTERS

Pointers are special variables that contain the *addresses* (the location in computer memory) of other variables. Any type of variable can have an associated pointer, including integers, characters, floating point numbers, arrays, and structures. Pointers may also be assigned to functions and procedures.

A generic pointer is also available. This type of pointer is used when you need a pointer but aren't sure what type of variable you will need to locate. A generic pointer is declared using the **void** variable type, like this:

```
void *ptr;
```

Keep in mind that array variables also function as pointers. For instance, the following demonstrates two ways to address the first character of a char array:

```
char *cp;
char name[10];

cp = &(name[0]);      /* the address of the first item */
cp = name;            /* the pointer to the first item */
```

Look at the memory diagram in Figure 6-1. This may give you a better idea of the relationship between arrays and pointers. It shows how memory would look while the following program (Program P6-4) runs, assuming that characters, integers, and pointers are all the same size.

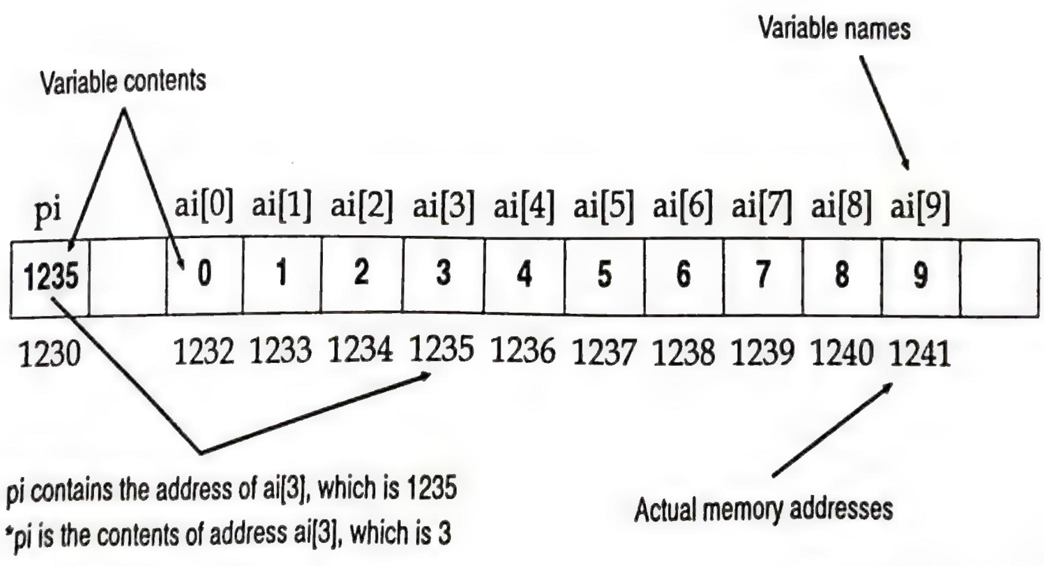


FIGURE 6-1
Pointers and memory

P6-4

```
/*
   C program to demonstrate pointers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

/* Declare global variables, outside of main() routine */
int *pi;    /* Pointer to an integer */
int ai[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
                 /* An array of integers, with preassigned
                    values */

main()                  "Address of"
{
     int i;
     pi = &ai[3];     /* pi points to the fourth element */
     for (i=0; i<5; i++)
         pi[i] = i;   /* pi is acting as an array now */
}
```

Pointer

Array notation

Pointers are declared by placing the * operator before the variable's name. So, to declare a pointer to an integer variable, use a declaration like this:

```
int *myVar;
```

Here the value of myVar is the address of the variable it is pointing to. If no value is currently assigned—that is, myVar == 0—the pointer is said to be a NULL pointer (pointing to nothing).

To access or change the value stored at the location pointed to by myVar, you need to use the * operator in a different way. For another look at the use of pointer variables, enter and run Program P6-5.

P6-5

```
/*
   C program to introduce pointers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   int i, j;        /* Two integer variables */
   int *pi, *pj;    /* Two pointers to integers */
```

```
                printf("Enter the first number: ");
                scanf("%d",&i);
Pointer ──────→ pi = &i;          /* pi is assigned the address ("&")
                                     of the variable i */
"Address of "
                *pi += 5;         /* Add 5 to the contents of the
                                     integer pointed to by pi (which is i) */
"Contents of "  printf("New value of i is %d\n",i);
                                  /* This will print the original i + 5,
                                     since pi points to the actual storage
                                     location of i */

                printf("Enter the second number: ");
                scanf("%d",&j);
                pj = &j;          /* pj points to the j variable */

                *pi = *pj;        /* This is the same as i = j, but uses
                                     pointer variables */

"Contents of "  printf("The contents of i: i=%d and *pi=%d\n",i,*pi);
                          /* Show i and the contents of the variable
                             pointed to by pi  */
                printf("The contents of j: j=%d and *pj=%d\n",j,*pj);
                          /* Show j and the contents of the variable
                             pointed to by pj  */

            }
```

Let's examine the **pi** and **pj** variables that are used as pointer variables in Program P6-5. (The asterisk in front of these variable names is not a part of the names; it merely denotes that the variables are pointers.) Since **pi** and **pj** are set to contain the addresses of i and j respectively, changing the values of i and j will change the values that ***pi** and ***pj** report. When referring to ***pi** and ***pj** within the code, the asterisk means "the contents of the variable pointed to by" **pi** or **pj**.

Note: The "address of" symbol (&) has been used throughout this book in examples having a **scanf()** statement. This is because **scanf()** requires a pointer to the values that are to be changed. You have already been using pointers, and didn't even realize it!

You may ask, "Why is there a different type of pointer for each variable type, if a pointer is just an address?". Actually, one type of pointer can be used to point to all types of variables. In these cases, a **cast** is used to let the compiler know what is going on. For instance, in the following example (Program P6-6), the p variable points to several different variable types:

P6-6

```
/*
   C program to demonstrate different types of pointers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>
char name[] = "Joe Smith";

main()
{
   int *p;          /* Our generic pointer */
   int  total, a, b;
   char *cp;        /* char pointer */

   p = &total;      /* First, p points to an int */

   printf("Enter a number: ");
   scanf("%d",&a);
   printf("How about another: ");
   scanf("%d",&b);

   total = a+b;
   printf("The total is %d\n",*p);
     /* Since p points to total, we can print this way. */
   cp = &(name[0]); /* cp points to the first char of name */
   p = (int *) cp;  /* p now points to the first char also */

   /* The next 2 statements print the same thing! */
   printf("The first character of name is %c (value %d)\n",
        *cp, *p);
   printf("The first character of name is %c (value %d)\n",
        *(char *)p, *(int *)cp);
}
```

Pointer

Address of

"Address of"

"Contents of"

Can be written as (char*) *p

Can be written as (int*) *cp

Before going on, take a look at the last two **printf()** statements in Program P6-6. Now examine the output from this program:

```
C:\CDT\CHAP6>p6-6
Enter a number: 12
```

```
How about another: 25
The total is 37
The first character of name is J (value 28490)
The first character of name is J (value 28490)

C:\CDT\CHAP6>
```

As a result of the first printf( ), the character printed by *cp is what you probably expected. The number printed by *p, however, may surprise you—it is the integer formed by the character byte printed by *cp, and the next byte (since integers require two bytes).

Similarly, in the second printf( ), *p is now cast to a character pointer, so only the single character byte is printed. And though *cp is now cast to an integer pointer, it prints the same number as before. This shows how important it is to make sure you declare your pointers to be of the correct type, or to be sure to cast them to the correct type when they are used.

In our examples so far, there has not been much reason to use pointer variables. When are they useful enough to warrant the extra work of declaring them? One common use for pointer variables is with array variables—especially strings, where you may not know the length of the the string but you need to check all the characters. Program P6-7 demonstrates this use of pointers.

P6-7

```
/*
   C program using string pointers.
   By: L. John Ribar
*/
#include <stdio.h>

main()
{
   char name[20];     /* A long word */
   char cn, cr;       /* New and replaced characters */
   char c;    Address /* A dummy character */
   char *cp;          /* Character pointer */

   printf("Please enter a long word: ");
   scanf(" %s",&name[0]);
    /* Point to the start of the string with &name[0]. */
                          "Address of"
   printf("Enter the char to replace and the new value: ");
   scanf(" %c %c",&cr,&cn);
      /* When you need to get two or more items from the
```

```
            user, use a scanf() with multiple specifiers.
            Remember, however, that the user needs to put a
            space between each parameter, not a comma, as is
            used in some other languages. */

    for (cp = &name[0];  *cp!=0;  cp++ )
       if (*cp == cr) *cp = cn;
             Contents
    printf("The new string is %s\n",name);

}
```

To bring all these new concepts about pointers into focus, let's examine, step by step, the following lines from Program P6-7:

```
for (cp = &name[0];  *cp!=0;  cp++ )
   if (*cp == cr) *cp = cn;
```

A standard **for** loop is used to step through the entire string, one character at a time. The first step is to initialize the index variable, **cp**; therefore, it is set to point to the first character in the string, with **cp = &name[0]**. Each time through the loop, ***cp!=0** is used to check for the end of the string. And, to move through the string, the pointer is incremented with **cp++**.

Within the loop, the character that **cp** points to (that is, the *contents* of **cp**, denoted ***cp**) is checked. If **cp** is pointing to a character with a value equal to **cr**, then the value is replaced with **cn**. The **for** loop condition could also be shortened to just ***cp**, rather than ***cp!=0**, because it returns a nonzero (true) value until it reaches the end of the string.

Note: Array variables are special, in that they are also pointers. In Program P6-7, the address of a character array was given as **&name**[0]. So the address of our name array can be given as **name**, without the address operator (&) or the array index ([0]).

ADDRESS ARITHMETIC

Look again at the following code from Program P6-7, and notice that the character pointer **cp** was incremented using the auto-increment (++) operation:

```
for (cp = &name[0];  *cp!=0;  cp++ )
   if (*cp == cr) *cp = cn;
```

What if the pointer is pointing to something that is more than one byte long? Do you have to increment the pointer for each byte? The answer to this is no; the compiler performs the calculations nicely.

One of the benefits of declaring a separate pointer for each type of variable is that during increment and decrement operations the compiler will add the correct amount to the pointer. Therefore, moving a pointer through a character array by bytes, through an integer array by words (two bytes), or through a floating point array by double words (four bytes each) is accomplished automatically when a pointer variable of the correct type is used with auto-increment (++) and auto-decrement (--) operators.

This feature of the C language improves the portability of C programs. Allowing the compiler to do the incrementing means you don't have to worry about the fact that the size of an integer variable may be different on various machines. Code from one machine will work with any other machine, as long as the variable sizes are handled by the compiler.

POINTERS TO ARRAYS

An array variable is actually a pointer to a location holding the first member of the array. If an array is defined using

```
int a[10];
```

then the variable **a** is actually treated like a pointer, pointing to the first of ten integer values. The actual members of the array are named **a[0]**, **a[1]**, and on up to **a[9]**.

Should you also wish to have arrays of pointers, or pointers to arrays, these are also possible. Enter and compile the following, Program P6-8. This program performs the same function as Program P6-3, but uses an array of pointers instead of a multidimensional array. In this case, several bytes of storage are saved, because all the "slots" for the month names are not preset to ten bytes each.

P6-8

```
/*
   C program to show more string examples.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

char *MonthName[12] = /* 12 month pointers */
   {
     "January", "February", "March", "April", "May", "June",
```

```
      "July","August","September","October","November","December"
    };

int days[] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };


main()
{
   int which = 0;  /* Initialize a simple variable */

   printf("\n Enter the number of the month: ");
   scanf(" %d",&which);
   while (which), /* If which = 0, this will be false */
   {
      printf("\n Month %d is called %s, and has %d days.\n",
                     which, MonthName[which-1], days[which-1]);
      printf("\n Enter the number of the month: ");
      scanf(" %d",&which);
   }
}
```

How about a pointer to an array? In the next example, Program P6-9, two arrays are created. Both contain the number of days in each month—one array is for regular years, and one is for leap years. The pointer variable is then set to the appropriate array, and values are available based on that decision.

P6-9

```
/*
   C program showing pointers to arrays.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

char MonthName[12][10] = /* 12 months, 10 characters each */
   {
     "January", "February", "March", "April", "May", "June",
     "July","August","September","October","November","December"
   };

int daysRegular[] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

int daysLeap[] =
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
typedef int BOOL;

main()
{
   int  which = 0;   /* Initialize a simple variable */
   int  year;
   BOOL by4, by100, by1000;
   int  *days;
          Pointer

   printf("Enter the month and the year (0 0 to quit): ");
   scanf(" %d %d",&which, &year);

   while (which)  /* If which = 0, this will be false */
   {
      by4=(year % 4) == 0;/* Is this evenly divisible by 4? */
      by100=(year % 100) == 0;    /* How about by 100? */
      by1000=(year % 1000) == 0; /* How about 1000? */

      if((by4)&&(!by100)) /* If divisible by 4 but not 100 */
      {
         days = daysLeap; /*
                                REMEMBER - this is the same as
                                using:
         Pointer     Array              days = &(daysLeap[0]);
                           */
      }
      else
      {
         if (by1000)      /* Check for millenia */
            days = daysLeap;
         else
            days = daysRegular;
      }

      printf("\n Month %d is called %s, and has %d days.\n",
                   which, MonthName[which-1], days[which-1]);

      /* Ask for another month to compute */
      printf("\n Enter the month and the year (0 0 to quit):
");
      scanf("%d %d",&which, &year);

   }
}
```

From Pascal to C

Pascal and C have many features in common. Although arrays, structures, and pointers are available in both languages, each language uses these elements quite differently. The following two programs, P6-10 and P6-11, perform the same functions, and demonstrate the differences between C and Pascal.

P6-10

```
/*
   C program to demonstrate arrays, structures, and
   pointers.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

typedef struct
{
   char FirstName[11];
   char LastName[16];
   int  Age;
} NameRecord;

/* Global variables */
NameRecord People[10];    /* 10 people */
NameRecord *aPerson;      /* Pointer to 1 person */
int        oneAge;        /* Assignment variable */

main()
{
   /* Assign values to a structure */
   People[1].Age = 10;

   /* Assign a pointer to be used with a structure */
   aPerson = &People[1];

   /* Now look at contents of the structure that is
      pointed to */
   oneAge = aPerson->Age;
}
```

From Pascal to C (*continued*)

P6-11

```
(*
   Pascal program to demonstrate arrays, records, and
   pointers.
   By: L. John Ribar, C DiskTutor
*)
PROGRAM P6_11;

TYPE
   NameRecord = RECORD
      FirstName : ARRAY[0..10] OF CHAR;
      LastName  : ARRAY[0..15] OF CHAR;
      Age       : INTEGER;
   END;  (* record *)

VAR
   People    : ARRAY[1..10] OF NameRecord; (* ten people *)
   aPerson   : ^NameRecord;      (* pointer to one person *)
   oneAge    : INTEGER;            (* assignment variable *)
BEGIN
   (* Assign values to a structure *)
   People[1].Age := 10;

   (* Assign a pointer to be used with a structure *)
   aPerson := ADDR(People[1]);
   aPerson := @People[1];   (* alternative method *)

   (* Now look at contents of the structure that is
pointed to *)
   oneAge := aPerson^.Age;
END.
```

Arrays and pointers are closely interrelated. Notice that our pointer variable in Program P6-9, declared as **int *days**, is used later in the **printf()** statement as an array variable, **days[which-1]**. Since an array variable is the same as a pointer to the first element in the array, these two variables can be used interchangeably.

BUILDING STRUCTURES

In contrast to arrays, which are groups of similar items, *structures* are groups of dissimilar items. For example, you might keep a list of phone numbers in an array. If you wanted to keep names and addresses with the phone numbers, however, you would need a structure.

SIMPLE STRUCTURES

A *structure* is used in C to define your own variable types that contain other variable types. A simple structure to hold a name and address might look like this:

```
struct
{
   char    *Name;
   char    *Address;
   char    *Phone;
} Info;
```

In a structure, the definition begins with the keyword **struct.** Inside curly braces is a list of variables that make up the structure. A variable name sometimes follows the list of *fields* (variables) and the last curly brace.

A field within the structure is accessed by using the structure name followed by a period and then the field name. Thus the following section of code assigns all the values of the structure:

```
struct
{
   char    *Name;
   char    *Address;
   char    *Phone;
} Info;

Info.Name = "Smith, Joe";
Info.Address = "123 West Main";
Info.Phone = "100-555-0000";
```

Notice that the **Name, Address,** and **Phone** variables were declared as character pointers because of how C handles strings. The string "Smith, Joe" is stored somewhere in memory by the compiler. To access it, you can assign a pointer to it, using

```
char *c;

c = "Smith, Joe";
```

Or you can use a function from the standard C library (see Chapter 11) to copy it to a local array of characters, like this:

```
char name[20];

strcpy( name, "Smith, Joe");
```

As with most other variables, structure variables may also be initialized when they are defined. The following code could replace the separate definition and assignment examples that you saw just above:

```
struct
{
   char *Name;
   char *Address;
   char *Phone;
} Info =
{
   "Smith, Joe",    /* first field */
   "123 West Main", /* second field */
   "100-555-0000"   /* third field */
};
```

When you define initial values for structures, you need not initialize all of the fields. For instance, suppose the foregoing example were written like this:

```
struct
{
   char *Name;
   char *Address;
   char *Phone;
} Info =
{
   "Smith, Joe",    /* first field */
   "123 West Main" /* second field */
    /* third field unknown */
};
```

In this case, only the first two fields would be initialized. The phone number would be filled in later in the program.

UNIONS

A data type similar to a structure is a *union*. In a structure, all the fields are available in a variable; the union structure, however, allows only one of the fields to be available at any one time. The union data structure can thus act differently at various times during the execution of a program.

The following union represents a unit of memory as used on IBM PC-compatible computers:

```
union
{
   char     bytes[4];
   int      words[2];
   long int bigword;
} PCaddress;
```

The layout of this union declaration is similar to that of a structure (except for the new keyword). The union, however, allows **PCaddress** (the space in computer memory) to be viewed in three different ways: as four characters, two integers, or as a single long integer. If you assign a value to any of the variables, the change occurs in the others also.

Load and run Program P6-12 now.

P6-12

```
/*
   C program to demonstrate unions.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

union MemStruct
{
   char byBytes[4];
   int  byInts[2];
   long byLong;
};
```

```
main()
{
   union MemStruct aMem;
   int i;  /* index variable */

   for (i=0; i<4; i++)
      aMem.byBytes[i] = i;    /* Assign values as bytes */

   printf("The bytes were %d, %d, %d, and %d\n",
      aMem.byBytes[0], aMem.byBytes[1],
      aMem.byBytes[2], aMem.byBytes[3] );
   printf("As ints, they are %d and %d, and %ld as a long.\n",
      aMem.byInts[0], aMem.byInts[1], aMem.byLong );

   aMem.byLong = 234987;    /* Assign a new value */
   printf("\nThe new values are:\n");
   printf("As bytes, they are %d, %d, %d, and %d\n",
      aMem.byBytes[0], aMem.byBytes[1],
      aMem.byBytes[2], aMem.byBytes[3] );
   printf("As ints, they are %d and %d, and %ld as a long.\n",
      aMem.byInts[0], aMem.byInts[1], aMem.byLong );

}
```

Here is the expected output from Program P6-12:

```
C:\CDT\CHAP6>p6-12
The bytes were 0,1,2, and 3
As ints, they are 256 and 770, and 50462976 as a long.

The new values are:
As bytes, they are 235, 149, 3, and 0
As ints, they are -27157 and 3, and 234987 as a long.

C:\CDT\CHAP6>
```

ARRAYS OF STRUCTURES AND UNIONS

Structures and unions work as other variables do. To create an array of structures and unions, place the size of the array in square brackets after the variable name. If several arrays are needed, however, or an assortment of single variables and arrays, you may wish to define the structure or union with a *tag*, (a name for the **struct** or **union**), which can then be used to create your own variable type.

Continuing with the name, address, and phone number example we've been using, you might create the following structure definition for use in other parts of the program. Notice that the tag value is placed before the field list, and the variable name after it.

```
struct AddrInfo
{
   char  Name[20];
   char  Street[20];
   char  City[30];
   char  Phone[15];
};
```

This structure does not create any variables; it just defines what the variables of this new type (AddrInfo) will look like. Now the structure can be used to define the necessary variables, like this:

```
struct AddrInfo Friends[100];
struct AddrInfo Employees[10];
struct AddrInfo ExWife;
```

You have thus far seen several variables that have similar types. Naturally, these definitions could have included pointers, as in:

```
struct AddrInfo *currentItem;
```

which can be used to move through the arrays. This is a good place to use a typedef statement. An improved version of the previous example follows:

```
struct AddrInfo
{
   char  Name[20];
   char  Street[20];
   char  City[30];
   char  Phone[15];
};

typedef struct AddrInfo ADDRESS;

ADDRESS Friends[100], Employees[10], ExWife, *thisItem;
```

This use of meaningful names in **typedef** statements makes the code much more readable, understandable, and simpler to debug. You can go one step further and combine **typedefs** with the **struct** definition, as shown here:

```
typedef struct AddrInfo
{
   char  Name[20];
   char  Street[20];
   char  City[30];
   char  Phone[15];
} ADDRESS;

ADDRESS Friends[100], Employees[10], ExWife, *thisItem;
```

POINTERS TO STRUCTURES

The process of defining pointers to structures is the same as for other variable types, but using the fields within structures works somewhat differently. Let's use the foregoing **typedef struct** example; you might assign **thisItem** to point to **Friends[0]**, the first friend in the list. To look at the **Name** field in the structure, you might assume you'd need a statement similar to this:

```
(*thisItem).Name
```

However, C has introduced a simpler way of expressing this type of access. Though a normal field in a structure is accessed using the period, a field under a structure pointer is accessed using the **->** operator. Thus, accessing the **Name** field is as simple as:

```
thisItem->Name;
```

Using the **->** operator is only necessary when the initial variable is a pointer. For instance, the next example, Program P6-13, demonstrates how subfields would be handled in each of these cases (pointer and nonpointer variables).

P6-13

```
/*
   C program to demonstrate subfields.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

struct date
{
   int  day;
   int  month;
   int  year;
};

typedef struct date DATE;

struct application                 Used as a new variable type
{
   char  name[20];
   DATE  today;
};

typedef struct application APPLY;

                  Structure     Pointer to structure
main()
{
   APPLY  original, *myCopy;  /* A struct and a pointer */
   char   c;                  /* To read the carriage
                                 return */

   printf("\n Enter your last name: ");
   scanf(" %s",&original.name);

   printf("\n Enter the day, month, and year for today: ");
   scanf(" %d %d %d",&original.today.day,
      &original.today.month, &original.today.year);
      /* Notice they all use periods */

   myCopy = &original;  /* Now use the pointer */

   printf("\n Your name was %s\n",myCopy->name);
   printf("The year is now %d\n",myCopy->today.year);
```

```
          /* In this case, myCopy points to the today field,
             but today has a direct subfield named year, so
             a period is used between today and year. */
}
```

In Program P6-13, when using the original variable, all fields and subfields are available using the period construct. When using the myCopy pointer variable, however, the fields are accessed using ->, and subfields need the initial -> along with a period.

Sometimes it is useful to declare a structure member to be a type that points to the structure type of which the member is a part. This is often helpful in linked lists (lists of items linked together using pointers), as demonstrated in the following, Program P6-14.

P6-14

```
/*
   C program to show a linked list.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

typedef struct list_item
{
    int datum;                          /* the data item */
    struct list_item *next;    /* pointer to the next one */
} LI;

LI item3 ={ 1, NULL };    /* The second variables are pointing */
LI item2 ={ 2, &item3 }; /* to the next item in the list. */
LI item1 ={ 3, &item2 }; /* NULL denotes the end of the list */

main()
{
    LI *item;    /* A pointer for moving through the list */

    for (item = &item1; item ; item = item->next)
        /* Start with the first item, and continue
           until we reach a NULL pointer */
        printf(" Data = %d \n", item->datum );
}
```

SUMMARY

- Arrays are collections of variables of the same type.
- Array elements are always numbered starting with 0, not with 1.
- Strings are character arrays. Strings always have a zero as the last element in the array, to mark the end of the string.
- Pointers are variables that hold the addresses of other variables.
- Pointers should be declared for the type of variable they will point to. In this way, the compiler can automatically update the pointer when using increment and decrement operations.
- Structures are collections of variables of different types.
- Unions are collections of various ways to look at the same place in memory.

MINI-CASE STUDIES

These mini-case studies will help you practice the concepts that you have learned so far. Example solutions are given in Chapter 12.

MINI-CASE 5

Develop a program called ENCODE.C that reads a string entered by the user from the keyboard, asks for a password number, and prints out an encoded copy of the string. If you use **scanf( )**, remember that this function stops reading at each space character; so for a long sentence it would be wise to enter it using periods instead of spaces. Encode the string by incrementing each character in the string by the number of characters given in the password number.

MINI-CASE 6

Develop a decoder program to go with the solution for Mini-Case 5. If you prefer, modify the ENCODE.C program from Mini-Case 5 to ask whether the user wants to encode (through adding the password) or decode (through subtracting the password).

C H A P T E R

PROGRAMS AND FUNCTIONS

In C, every program contains one or more *functions*—these functions are the workhorses of the C program. You have already learned that every C program has at least one function, always called **main( )**, which is called when the program starts. But **main( )** is usually not the only function within a program.

A function is a group of C statements that are executed together. In some languages (such as Fortran), you use a CALL statement to call a function. In C, however, just listing the function's name (followed by a pair of parentheses) starts the function executing. *Parameters* are the values or variables passed (or sent) into the function. The parameters are placed between the parentheses when the function is called.

Functions have several uses. They help break your code into smaller, more understandable pieces. Also, by having a sequence of statements grouped in a function, other programs or even other functions can reuse the code in the function.

Tip: When you design a function, have a single purpose in mind for that function. The function may or may not have inputs and outputs, and it may be written to have no interaction at all with the calling

routine. The function must, however, have a purpose; this purpose should be determined in advance, and then followed as the code is written.

In this chapter, you will learn about the structure of programs and functions. You will see how *prototypes* are used as a means of documenting the functions available to a program. You will learn about functions that retrieve information from the command line and from the operating environment for use in the program. Finally, you will study all these concepts in an example containing a function called **scan( )**.

DEVELOPING YOUR PROGRAM'S STRUCTURE

Before you write a program, you need to develop a design for it—you learned about basic program design and style in Chapter 2. This program design is a road map for writing the program itself. Remember that the design process should answer the following questions:

- What will the program do?
- What input, or data, is required from the user of the program?
- What processes, or functions, need to be performed on the data?
- How will the processes that need to be performed be translated into C functions?
- What output is expected?

Once the program design is complete, you can begin the coding. For documentation purposes, and as part of your programming style (described in Chapter 2), try to lay out your program files in a consistent form, in sections, something like this:

```
/* Comments about the program, the author, date, etc. */


/* Preprocessor items  */


/* Structure Definitions  */
```

```
/*  Static and Global Variables  */


/*  Function Prototypes  */


/*  Functions  */
```

In the *Comments* section of a program, write a comment to describe the purpose of the program. If the file does not contain a complete program, this comment can explain the purpose of the functions contained in the program segment. Comments will vary widely, of course, from one programmer to the next, but should always include at least a statement of purpose, the author's name, and the date the file was last edited.

Preprocessor items (which will be covered in more detail in Chapter 11) will include definitions used within the file, a list of other files to include, and conditional statements to control the parts of the code that will be compiled under certain circumstances.

Structures used within the C file, but not defined in any of the included header files, should be described at the top of the file, so that another programmer reading the code will immediately know the design of the structures.

Now let's begin our examination of the variables and functions that are so important to your program's execution.

USING VARIABLES AND FUNCTIONS IN YOUR PROGRAM

There are two types of variables: *automatic* (also called *global* or *dynamic*) and *static.*

Automatic variables declared outside of any function are available from all functions within your program. They can be accessed from other source files by declaring them as *external* (that is, they are external to the current source file). This is done by placing the keyword **extern** before the variable declaration, like this:

```
extern int i;
```

In this case, **i** is declared as an integer that exists externally to the current file. This declaration does not reserve any space for the variable; it just says that the variable is assumed to exist elsewhere.

In contrast, static variables are only available to functions within the source file where the variable is declared. These variables are declared *outside* of any function, and have the keyword **static** placed before their name.

It is also possible to have static variables *within* a function. Normally, variables declared within a function lose their values between calls to that function, and so you need to initialize the variables during each call of the function. With static variables, however, the variables hold their values between calls to the function. (These values remain unavailable outside of the function itself.)

Program P7-1 contains an example of static variables. (Don't worry about the prototype section of this program; you'll learn about prototypes later in this chapter.)

P7-1

```
/*
   Program Name:  P7-1.C

   Purpose:
   This program was written for the C DiskTutor. This is an
   example of a small program with static variables inside and
   outside functions.

   Author:        L. John Ribar

   Last Update:   12/27/92
*/

/*  Preprocessor Items  */

#include <stdio.h>

/* Two function prototypes are defined first.*/

int CountStatic( void );
int CountNonStatic( void );

/* Static and Global Variables */

static int top = 0;
```

Prototypes

main() is first

```
/* Main Program Code */

main()
{
   int i;

   printf("top is %d\n",top);
   for (i=0; i<5; i++)
   {
      top ++;
      printf("top = %d, static count = %d, dynamic count =
            %d\n", top, CountStatic(), CountNonStatic());
   }
}

/* Now provide the functions prototyped above.*/

/*
   This function counts, using a static variable.
*/
int CountStatic( void )
{
   static int i = 0;   /* Declare and initialize the static
                          variable */

   return( ++i );
}

/*
   This function counts, using a nonstatic variable.
*/
int CountNonStatic( void )
{
   int i = 0;   /* Declare and initialize the dynamic
                   variable */
   return( ++i );
}
```

Other functions follow main()

void indicates that no parameters are used

Examine the output of Program P7-1, which is shown here:

```
C:\CDT\CHAP7>p7-1
top is 0
top = 1, static count = 1, dynamic count = 1
top = 2, static count = 2, dynamic count = 1
top = 3, static count = 3, dynamic count = 1
top = 4, static count = 4, dynamic count = 1
```

```
top = 5, static count = 5, dynamic count = 1

C:\CDT\CHAP7>
```

Notice that the static counting routine **CountStatic( )** returns increasing numbers, since the **i** variable is only initialized the first time; the function retains the value of the **i** variable between calls to that function. The function **CountNonStatic( )**, on the other hand, resets the initial value each time that function is called.

Note: As usual, the first function in these C programs is the **main( )** function. This is different from many other languages such as Pascal, in which the main function is placed at the end of the file.

Just as there are static variables, there are also *static functions*. These are functions that can only be called from other functions within the same program file.

Note: All variables have a **scope** in which they are valid. The **scope** of a variable is the part of the program in which it is valid to use that variable. A variable declared within a function is only available within that function. A variable declared outside all functions is available within all functions, unless declared with the **static** keyword. Static variables declared outside a function are available only to the functions within the same source file.

In the next example, Program P7-2, you will see several functions used, some with and some without parameters.

P7-2

```
/*
    Program Name:  P7-2

    Purpose:
    This is an example of a program with several functions.

    Author:        L. John Ribar

    Last Update:   12/27/92
*/
```

```
/*  Preprocessor Items  */

#define Yes 1       /* Replace occurrences of Yes with 1 */
#define No 0        /* and all occurrences of No with 0 */

#include <stdio.h>


/*  Structure Definitions  */

typedef int BOOL;  /* Define a BOOLean type */

struct date         /* Define a date structure */
{
   int month;
   int day;
   int year;
};

typedef struct date DATE;  /* Rename the date type */

/*  Static Variables  */

static struct date Today;

static int DaysInMonth[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

static BOOL changedMonth;

/*  Function Prototypes  */

int add2( int );    /* prototype */

static void add15days( struct date * );  /* prototype */


void main( )
{

   printf("This is a demo program with nice structure.\n");
   printf("We will assign a date of 5/20/92. Fifteen\n");
   printf("days after that will be ");
```

```
    Today.month = 5;
    Today.day = 20;
    Today.year = 92;
    add15days( &Today );
    printf("%2d/%2d/%2d \n",Today.month, Today.day,
          Today.year);

    if (changedMonth)
       printf("This is in a new month.\n");

    printf(" If you add 2 to 15 you get %d\n",add2(15));
}


/*
   This function adds 2 to a number and returns the total.
*/
int add2( int number )
{
   return (number+2);
}


/*
   This routine adds 15 days to the date sent in. It
   corrects for month overrun based on the length of the
   month. Since it is static, no other files may call this
   routine.
*/
static void add15days( struct date *thisDate )
{
   int d, m;

   changedMonth = No;

   d = thisDate->day;
   m = thisDate->month;

   if (d+15 > DaysInMonth[m])
   {
      if (m==12) /* December */
         thisDate->month = 1;
      else
         thisDate->month++;  /* Move to next month */
      thisDate->day = (d + 15) - DaysInMonth[m];
```

```
        changedMonth = Yes;
    }
    else
    {
        thisDate->day += 15;
    }
}
```

Program P7-2 introduces several new concepts:

- Using a return value from a function in another statement
- Using static and dynamic variables
- Using preprocessor directives

First, in the last **printf( )** statement within **main( )**, the return value for the function **add2( )** is used as the data for the **%d** designator. This shows how the return value of a function is valid for use whenever a variable of the same type is expected.

Static variables are declared at the top of the file. Remember—static variables retain their values no matter how long the program runs. Also, no functions outside of this file may access these variables directly. And functions, too, can be defined as static; this was done with **add15days( )** in the Function Prototypes section, so this function can only be called by other functions within the same file.

Dynamic variables are created when a function is called, and then destroyed when the function completes. These variables are declared within the function, before any code. In the example above, the **d** and **m** variables in **add15days( )** are both dynamic. Since dynamic variables do not hold their values between calls of a function, they should be initialized each time the function is called.

Several *preprocessor directives* are used in Program P7-2. (See Chapter 11 for more coverage of preprocessor directives.) You've seen **#include** previously; it instructs the compiler to load the contents of the file listed. The **#define** statement is used to define values that will be substituted in the final compilation. For instance, whenever the word **Yes** is seen in the code, it will be replaced by the number 1 when the compiler is working. This type of **#define** statement is very useful in making code easier to read.

Program P7-2 also includes the *prototypes* for the two functions in the program. Prototypes are discussed next.

UNDERSTANDING FUNCTIONS AND PROTOTYPES

The functions you use in your C programs will have the same general format as the **main()** function. There is a function name, a possible *return value,* and zero or more parameters. A block of C code follows, within curly braces. Here are several examples of functions:

```
int func1 ( void )
{
   /* code */
}


char *func2( char ch )
{
   /* code */
}


void func3( int *i )
{
   /* code */
}


void func4( float f, int t )
{
   /* code */
}
```

In the above examples:

- **func1()** has no parameters, denoted by **void** in the parentheses. This function does, however, return an integer value, by means of the **int** keyword preceding the name of the function.
- **func2()** takes a single character as a parameter, and returns a pointer to a character. This is denoted by the **char** keyword followed by an asterisk before the function name.
- **func3()** and **func4()** do not return any values (note the **void** before the function names). **func3()** takes a single parameter—a pointer to

an integer; func4( ) takes two parameters—a floating point value and an integer value.

You might guess correctly that there are nearly endless ways to define functions. Many different parameters may be passed in a function call, and different types of variables may be used as parameters and return values.

WRITING FUNCTION PROTOTYPES

Before you enter a function call in your program, you should define a prototype. A *prototype* lets the compiler examine the variable types being sent to a function and determine any possible conflicts. The prototype also documents the interface used to call the function, in case it needs to be used again.

The prototype statement consists of the first line of the function, followed by a semicolon instead of the curly braces. You should define the function's prototype *before* any calls to that function are made in the program.

Here are the prototypes for the three functions you examined just above:

```
int func1 ( void );

char *func2( char ch );

void func3( int *i );

void func4( float f, int t );
```

PASSING PARAMETERS TO FUNCTIONS

Passing parameters into C functions is a fairly straightforward programming task. The type of the parameter and the parameter's name are displayed in the function prototype, as shown in the previous examples. When your program calls for the parameters to be modified within the function, however, then some special rules must be followed.

Normally, the call to a C function passes copies of most of the parameters, including integers, characters, floating point numbers, doubles, structures, and unions. Because of this, if the variable will be changed within the function, a pointer to the variable (its address in memory) must be passed instead. Look at Program P7-3 now, as this discussion continues.

P7-3

```
/*
   Program Name:  P7-3

   Purpose:
   This C program shows examples of functions with parameters.

   Author:        L. John Ribar

   Last Update:   12/27/92
*/


/*  Preprocessor Items  */

#include <stdio.h>

/*  Prototypes  */

void TryToAdd( int i, int j );
void DoAdd( int *i, int j );

/*  Main Program Code  */

main()
{
   int i;

   printf("This program uses two function calls to\n");
   printf("demonstrate what happens when a pointer\n");
   printf("is not used in a function call.\n");

   i = 5;
   printf("i starts with a value of %d\n", i);

   TryToAdd(i,5);
   printf("After TryToAdd, i is %d\n", i);

   DoAdd(&i,5);
   printf("After DoAdd, i is %d\n", i);

}

*
   This function tries to add two numbers, but cannot return
   the result because no pointer is available.
```

```
*/
void TryToAdd( int i, int j )
{
    i += j;
}


/*
   This function adds two numbers together, returning the
   result in the first parameter.
*/
void DoAdd( int *i, int j )
{
    *+= j;
```

The TryToAdd( ) function accepts two integer parameters. Since these are sent as copies, no change is made to i when it returns. The DoAdd( ) function accepts i as a pointer, so the function changes the actual value of i. This is because C needs the actual location of the variable (the address of i, or a pointer to i) to know where to store the result. Run the program now to see how the results (shown here) prove this conclusion.

```
C:\CDT\CHAP7>p7-3
This program uses two function calls to
demonstrate what happens when a pointer
is not used in a function call.
i starts with a value of 5
After TryToAdd, i is 5
After DoAdd, i is 10

C:\CDT\CHAP7>
```

Notice that when DoAdd( ) is called, i needs the & operator to provide an address. If i were a pointer variable, this operation would not be required. Also, notice that within the DoAdd( ) function, i is considered to be a pointer, and therefore the * operator is used to indicate that the contents of the variable pointed to by i is what should be changed. Without this operation, the pointer itself would have been changed.

Remember: An array variable name is actually a pointer to the first element in the array. Therefore, the & operator is not needed before the variable name in the calling sequence.

From Pascal to C

C and Pascal are quite similar in how they deal with functions, parameter passing, and return values. There are also a few major differences. Programs P7-4 and P7-5 show some of these similarities and differences.

P7-4

```
/*
   Program:      P7-4
   By:           L. John Ribar, C DiskTutor
   Date:         December 27, 1992
*/

/*
   Function that returns an integer variable, twice the
   number passed in.
*/
int DoubleIt( int i )
{
   return i*2;
}
              The return value
/*
   Function that squares the number passed in, and
   returns it in the same variable.
*/
void SquareIt( int *i )
{
   *i = (*i) * (*i);  /* The parentheses around *i are
                         not necessary, but they make
                         reading the multiplication
                         statement simpler. */
}

main()
{
   int aNumber;

   aNumber = DoubleIt( 5 );
   SquareIt( &aNumber );
}
```

From Pascal to C (*continued*)

P7-5

```
PROGRAM P7_5;
VAR
   aNumber : INTEGER;

(*
   Function that returns a number twice the number
   passed in.
*)
FUNCTION DoubleIt( i : INTEGER ) : INTEGER;
BEGIN
   DoubleIt := i * 2;
END;

(*
   Procedure that squares the number passed in, and
   returns it in the same variable.
*)
PROCEDURE SquareIt( VAR i : INTEGER );
BEGIN
   i := i * i;
END;

BEGIN
   aNumber := DoubleIt( 5 );
   SquareIt( aNumber );
END.
```

Different types of calls

The return value

In Pascal, unlike C, a function that returns a value is called a FUNCTION, and a function that does not return a value is called a PROCEDURE. Also, Pascal uses a VAR parameter to pass a variable that will be changed. Using VAR with a parameter causes Pascal to pass a pointer, as C does, but it is done transparently to the programmer. Thus Pascal programmers do not have to use pointer variables as often within their functions and procedures.

PROGRAMS THAT READ COMMAND-LINE AND ENVIRONMENT OPTIONS

Thus far, the **main()** function has always been shown without any parameters. However, **main()** can also be used with parameters that make DOS command-line parameters, and operating environment settings, available to the C program.

To read the command line, you can use this **main()** function prototype:

```
main( int argc, char *argv[] );
```

which provides two parameters for **main()**. The **argc** parameter is a number that represents how many items were found on the command line. And **argv** is an array of pointers that point to the actual command-line parameters (each of which is an array of characters).

Program P7-6 shows how these parameters can be used within a program.

P7-6

```
/*
   Program Name:  P7-6

   Purpose:
   This is an example of a program that displays the arguments
   passed on the command line.

   Author:        L. John Ribar

   Last Update:   12/27/92
*/


/*  Preprocessor Items  */

#include <stdio.h>


/* Main Program Code */

main( int argc, char *argv[] )
{
   int i;
```

```
    if (argc < 2)
       printf("No arguments given!\n");
    else
       for (i=0;  i<argc;  i++)
          printf("Argument %d is %s\n",i,argv[i]);

}
```

Run Program P7-6 several times with different command-line arguments (as shown below), to see how this works.

```
C:\CDT\CHAP7>p7-6
No arguments given!
```

Nothing followed the program name

```
C:\CDT\CHAP7>p7-6 Hello there! I'm here now!
Argument 0 is C:\CDT\CHAP7\P7-6.EXE
Argument 1 is Hello
Argument 2 is there!
Argument 3 is I'm
Argument 4 is here
Argument 5 is now!
```

Command-line parameters

```
C:\CDT\CHAP7>
```

This method of reading the command line (parameters for the **main()** function) gives the programmer a powerful method of interfacing with the user. Instead of requesting several items from the user when the program starts, the program can read the command line, and use the command-line arguments automatically.

Notice that Program P7-6 looks for two arguments on the command line (**if (argc < 2)**) before continuing. This is necessary because of how compilers handle the command line. It is common practice to reserve **argv[0]** to hold the program name. In operating environments where this is not done, **argv[0][0]** is actually NULL (zero). Thus if there is only one command-line argument (**argv[0]**, the name of the program), there is nothing to be done in this program.

Another powerful method of interfacing with the user is with *environment strings*. The operating environment contains information your program may need in order to execute. For instance, under MS-DOS, the environment usually contains the current path, the current system prompt, and other variables created with the SET statement. To access these variables, a C

program can call the **getenv( )** function. The prototype for this function is in the stdlib.h file, which must then be included in the C program.

The **getenv( )** function returns a pointer to a character string (array) containing the value of the variable. If the variable is not found in the environment, a NULL pointer is returned. Here is the prototype for getenv( ):

```
char *getenv( const char *name );
```

Do not change the value of the string returned by the call, as this is a constant value held by the system during program execution. The **const** keyword denotes that although a pointer is being passed to this routine, **getenv( )** will not change it.

Run Program P7-7 several times, to see in action the routines that access the command line and environment options. Here is the program and its results:

```
/*
   Program Name:  P7-7

   Purpose:
   This is an example of a program that accesses the command
   line and the environment strings.

   Author:        L. John Ribar

   Last Update:   12/27/92
*/


/*  Preprocessor Items  */

#include <stdio.h>
#include <stdlib.h>

/*  Main Program Code  */

main ( int argc, char *argv[] )
{
   char *value;

   if (argc < 2)
   {
```

```
        printf("Please use a command-line argument that\n");
        printf("represents an environment variable, like\n");
        printf("PATH or PROMPT\n");
    }
    else
    {
        value = getenv(argv[1]);
/*          NULL is defined in stdio.h as having a value of 0.
            It is often used to check for empty pointers and
            empty strings.*/
        if (value == NULL)
            printf("That variable was not located.\n");
        else
            printf("The value of %s is %s\n",argv[1],value);
    }
}
```

```
C:\CDT\CHAP7>p7-7
Please use a command-line argument that       <--- Help message if no
represents an environment variable, like           arguments
PATH or PROMPT

C:\CDT\CHAP7>p7-7 prompt
The value of prompt is $P$G                   These values are in
                                              the environment
C:\CDT\CHAP7>p7-7 temp
The value of temp is C:\TEMP

C:\CDT\CHAP7>p7-7 other
That variable was not located. <---- This one was not found

C:\CDT\CHAP7>
```

AN EXAMPLE OF A FUNCTION IN ACTION

This practical example, Program P7-8, demonstrates a function that might be useful in Case Study 1 in Chapter 8. The function, called scan( ), reads input from the user one character at a time. As the characters are read, the total number of characters is counted, as well as the number of

words entered (computed from the number of times the SPACEBAR is pressed).

The prototype for the **scan( )** function is

```
int scan( char *input, int *words );
```

The length of the string typed at the keyboard is returned by **scan( )**. The actual string entered is placed in **input** and can be parsed into separate words by your main routine, using a function like **sscanf( )**, described in Chapter 9.

Notice the preprocessor directives **#ifdef**, **#define**, and **#endif**. These are used to compile portions of code based on conditions. In this example, some code is included at the bottom of the file to test the **scan( )** function. Once this routine works, if you wish to use it in your own code, just remove the line of the program that says

```
#define MAIN
```

and the test code at the bottom will no longer be included when the program is executed.

Here is Program P7-8, the SCAN program:

P7-8

```
/*
   File Name:  SCAN.C       (a.k.a. P7-8.C)

   Purpose:
   This file contains a scanning function that is used to read
   input from the user, counting the length of the string
   entered and the number of words (spaces) entered. In
   addition, if two or more spaces in a row are entered,
   they are ignored.

   Author:         L. John Ribar, C DiskTutor

   Last Update:    12/27/92
*/


/*  Preprocessor Items  */

#include <stdio.h>
#include <conio.h>
```

```
/*
   The scan function is used to retrieve input from the user
   while giving the programmer better control over what is
   read. There are three pieces of information passed back:

      char *input         The line of data entered
      int *words          The number of items entered
      returned int        The length of the input line
                          0 means the person just pressed the
                          carriage return.

   Pseudocode:

      Get a Char
      WHILE Char != RETURN
         Ignore Extra Spaces
         Store the Char
         If the Char is a Space, Increment # of Words
         Increment Length
         Get a Char
      END;
*/

int scan( char *input, int *words )
{
   int pos, len;
   char c;

   *words = 1;     /* Number of words entered */
   pos = 0;        /* Storage location in input array, and
                      length of what has been typed so far */

   c = getche();    /* Read a char from the user */
   while ( c != 13 )
   {
      /* First strip off leading spaces, or duplicate spaces
         within the input stream.*/
      if (
           (c==' ') &&
           ( (input[pos-1]==' ') || pos==0 )
         )
         ;  /* Do nothing - ignore the space.*/
      else
      {
```

```
            input[pos] = c;  /* Store the character pressed.*/
            pos++;           /* Get ready for the next
                                character.*/
            if (c == ' ')
               (*words)++;   /* If it was a space, increment the
                                number of words read this far. */
         }
         c = getche();       /* Get another character.*/
      }
      input[pos] = 0;    /* End all C strings with a zero! */
      if (pos == 0)
         *words = 0;
      printf("\n");
      return pos;
   }

   /*  Test code follows  */
   #define MAIN

   #ifdef MAIN

   void main()
   {
      char string[80];
      int words;
      int len;

      printf("Trying out scan routine.\n");
      string[0] = 0;
      while (string[0] != 'q')
      {
         len = scan( string, &words );
         printf("You entered %d words, and ", words);
         printf("%d characters\n<<%s>>\n",
               len, string );
      }
   }

   #endif
```

And here is a sample run of the SCAN program:

```
C:\CDT\CHAP7>P7-8
Trying out scan routine.

You entered 0 words, and 0 characters
<<>>
On the last line, I just entered the Return key. . .
You entered 10 words, and 50 characters
<<On the last line, I just entered the Return key. . .>>
Goodbye for now!
You entered 3 words, and 16 characters
<<Goodbye for now!>>
Quit
You entered 1 words, and 4 characters
<<Quit>>
q
You entered 1 words, and 1 characters
<<q>>

C:\CDT\CHAP7>
```

User has just pressed ENTER

Sentences

Reports from scan() function

Remember, once you are satisfied with the results of this program, remove this line:

```
#define MAIN
```

and this scan( ) will be ready for use in your own programs.

SUMMARY

- Every C program has a function called main( ).
- Design your functions to perform a single task.
- Before writing a program, design it, following established program design steps.
- Variables can be global or static. Global variables are accessed from other files using the extern keyword. Static variables are not accessible from other source files.

- Functions can also be declared as static.
- Parameters must be passed into functions as pointers if you want to change the values of the parameters.
- Functions cannot be called until you define their prototypes.
- Command-line arguments are read using parameters in the **main()** function call.
- Environment variables are read using the **getenv()** function.
- Use the **scan()** function (the SCAN program, numbered P7-8) presented in this chapter to make your own programs read user input.

MINI-CASE STUDIES

The following mini-cases help you to design programs that read the command line, use functions, and manipulate structures.

MINI-CASE 7

Develop a program called WHENNEXT that reads the name of a day (Monday, Tuesday, Wednesday, and so on) from the command line, and displays the date of the month for the next occurrence of that day. For instance, if today were Friday, November 13, and you ran this program:

```
WHENNEXT Monday
```

the expected result would be

```
Monday, November 16
```

You will need to use two functions from the system library, whose prototypes are shown here:

```
#include <time.h>

time_t time( time_t *tloc );
struct tm *localtime( const time_t *timer );
```

MINI-CASE STUDIES (*continued*)

These functions, and the variable types they use, are both declared in the header file <time.h>. Be sure to include this header in your program.

The **time()** function is used to get the current time, in encoded format, into a variable declared with type **time_t**. The **localtime()** function then converts this encoded time into a **struct**, which is declared as follows:

```
struct tm {
    int  tm_sec;     /* seconds after the minute -- [0,61] */
    int  tm_min;     /* minutes after the hour   -- [0,59] */
    int  tm_hour;    /* hours after midnight     -- [0,23] */
    int  tm_mday;    /* day of the month         -- [1,31] */
    int  tm_mon;     /* months since January     -- [0,11] */
    int  tm_year;    /* years since 1900                   */
    int  tm_wday;    /* days since Sunday        -- [0,6]  */
    int  tm_yday;    /* days since January 1     -- [0,365]*/
    int  tm_isdst;   /* Daylight Savings Time flag */
};
```

Since **localtime()** returns a pointer to a structure **tm**, you will need a pointer variable to access the information. A program to display the current time would look like the following, Program P7-9, called NOW.

P7-9

```
/*
   Program:     NOW.C     (a.k.a. P7-9.C)

   Purpose:     Print the current time.

   By:          L. John Ribar, C DiskTutor
                December 27, 1992
*/
#include <stdio.h>
#include <time.h>

main()
{
   time_t nowEncoded;
   struct tm *Now;
```

MINI-CASE STUDIES (*continued*)

```
    nowEncoded = time( NULL );
     Now = localtime( &nowEncoded );
     printf("It is now %d:%02d\n", Now->tm_hour, Now->tm_min );
}
```

MINI-CASE 8

Change the foregoing NOW program so that it also prints whether the current time is A.M. or P.M. Instead of

```
It is now 14:22
```

you want the program to print

```
It is now 2:22 p.m.
```

C H A P T E R 8

DETAILED EXAMPLE: A SCREEN LIBRARY

Now that you have completed seven chapters of this book, you have learned a great deal about the C language. Here in Chapter 8 you will be able to watch the entire development process, including planning and documentation, of a full C program.

> **Tip:** Take your time as you study the detailed example in this chapter. The functions developed here demonstrate most of the concepts covered so far in this book. Also, this code will be very useful in the case studies, and in your own programs.

The detailed example in this chapter is built around a library of useful screen-handling tools for use by your own C programs. There are two parts to the project: First, a set of screen-handling functions are developed. Once these are complete, functions to provide your own programs with a *text-based user interface* (TUI), as shown in Figure 8-1, are developed. These functions will form the core of the Screen Library.

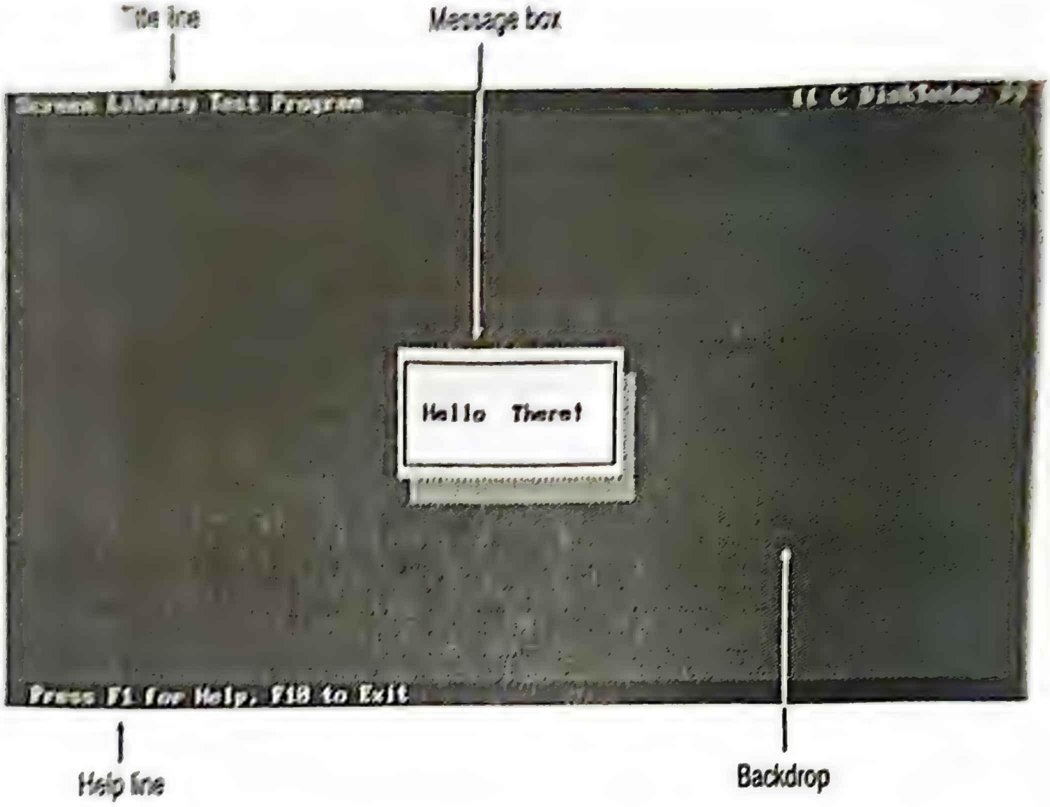


FIGURE 8-1
A text-based user interface (TUI) screen

Note: A *library* is one or more functions specifically developed to be useful for more than just one program. This type of code is generally kept in a separate C source file, and linked with new programs as they are written. This saves you from having to rewrite commonly used functions.

THE DESIGN SPECIFICATION

Because there are two parts to your Screen Library, the design document for the library also has to identify two sets of functions: the screen-

handling functions, and the text-based user interface (TUI) functions. Let's take a look at the components of each part.

SCREEN-HANDLING FUNCTIONS

The library must perform the following functions:

- Clear the screen.
- Move the cursor to a specific location on the screen.
- Print text on the screen at a particular location.
- Print text on the screen in a specific color.
- Determine if any keys have been pressed by the user.
- Read keystrokes entered by the user from the keyboard, and recognize special characters such as function keys and arrow keys.
- Scroll a portion of the screen up or down. This means whatever is in that area of the screen is shifted up or down a specified number of rows.
- Draw a shadow around a given area of the screen. A *shadow* is a darkened area at the right and bottom sides of a specified screen area, giving a three-dimensional look to the area. Look at the shadow around the box in Figure 8-1.

In addition to the specific functions listed above, the code must support two types of screen interfaces:

- *ANSI screen functions.* These are shown in the second part of Appendix B and are supported by most computers and terminals.
- *PC BIOS screen functions.* These are faster and more versatile than the ANSI functions, but are available only on IBM PC-compatible computers. (For a thorough explanation of BIOS functions, refer to one of the PC programming books listed in the Bibliography.)

TEXT-BASED USER INTERFACE (TUI) FUNCTIONS

Once the basic screen-handling functions are implemented, the library will be extended to perform the following TUI functions. (Figure 8-1 shows you the result of these functions.)

- Draw a *backdrop* on the screen. A backdrop is a background that fills the entire screen and gives the interface a professional appearance.
- Place a title at the top of the backdrop screen, and allow changing of the title without extensive work.
- Place help text on the bottom row of the backdrop screen, and allow changing of the help text without extensive work.
- Draw a box on the screen. Optionally, fill the box with space characters and/or add a shadow around the box.
- Create a message-box function, which will display a given text string in a shadowed box in the center of the screen. Allow removal of the box by filling the box area with backdrop characters.

CREATING A SCREEN LIBRARY

To keep the code simple, and to provide maximum flexibility in the finished product, the functions required by the design document will be written as separate C functions. These functions will also become the tools needed to build the TUI functions.

PROTOTYPES FOR THE SCREEN-HANDLING FUNCTIONS

Here is an explanation of the prototypes for the screen-handling functions:

```
void Move( unsigned row, unsigned col );

void Where( unsigned *row, unsigned *col );

void Say( unsigned row, unsigned col, char *text );

void SayColor( unsigned row, unsigned col, char *text,
               char attr );

char Inkey( void );

int Keypressed( void );

void clrscr( void );
```

```
void Shadow( int row1, int col1, int row2, int col2,
             char attr );

void Scroll( unsigned row1, unsigned col1, unsigned row2,
             unsigned col2, int moves, char Att );
```

Note: The screen interface functions listed here work with a standard PC screen, 80 columns wide and 25 rows high. Rows and columns start from the upper-left corner as row 0, column 0, and increase to row 24, column 79 as they move to the bottom-right of the screen.

The **Move( )** function places the cursor at the requested position on the screen. **Where( )** determines the current location of the cursor on the screen. **Say( )** displays given text at a specified location, and **SayColor( )** does the same thing using a given color for the text. **Inkey( )** returns the key the user pressed from the keyboard. **Keypressed( )** returns True (1) if the user has pressed a key, and False (0) if the user has not. The **clrscr( )** function clears the screen. **Shadow( )** places a shadow on the bottom and right sides of the screen area specified. **Scroll( )** scrolls the given area up or down a specified number of rows.

In addition to the foregoing functions, two utility functions are provided for screen handling. These functions convert foreground and background colors into a single attribute byte (required by BIOS), and convert an attribute byte into two separate foreground and background colors (required by ANSI). Here are the prototypes for these utility functions:

```
char Attr( int fore, int back );

void ColorsOf( char attr, int *fore, int *back );
```

PROTOTYPES FOR THE TUI FUNCTIONS

Now let's look at the prototypes for the TUI functions:

```
void ResetArea( int r1, int c1, int r2, int c2, char ch, char
                attr);

void Backdrop( char ch, char attr );

void ReplaceBack( int row1, int col1, int row2, int col2 );
```

```
void DrawBox( int row1, int col1, int row2, int col2,
              int vert, int horiz, char attr, int fill, int
              shadow );

void HelpSetup( unsigned row, char attr );

void ClearHelp( void );       /* Clears help message */

void SayHelp( char *txt );

void TitleSetup( unsigned row, char attr );

void ClearTitle( void );

void SayTitle( char *txt );

void MessageBox( char *text );

void ClearMessageBox( void );
```

The ResetArea( ) function fills an area of the screen with a specified character and attribute (color). Backdrop( ) draws the background on the screen. ReplaceBack( ) replaces a portion of the background that has been covered by a box. DrawBox( ) draws a box on the screen, using the given color; optionally, the box can be filled in, and/or a shadow can be added.

HelpSetup( ) and TitleSetup( ) are used to specify the row and color used with all the Help and Title functions—that is, the functions related to the title line and help line on the screen. If the row for either line is specified as -1, then that line is disabled and will not appear on the screen. To make this simpler, these two #define statements are placed in the Screen.H header file (you'll learn about header files in the section immediately to follow):

```
#define NO_HELP (-1)
#define NO_TITLE (-1)
```

With these defined, you can disable the help and title lines by calling HelpSetup( ) and TitleSetup( ) as follows:

```
HelpSetup( NO_HELP, 0 );
TitleSetup( NO_TITLE, 0 );
```

ClearHelp( ) and ClearTitle( ) clear out the help and title displays on the screen, and SayHelp( ) and SayTitle( ) put new text into these lines.

MessageBox( ) displays a message in a shadowed box in the center of the screen. ClearMessageBox( ) replaces the background behind the message box, effectively erasing the message box from the screen.

THE HEADER FILES

All function prototypes are placed in a header file called Screen.H. To allow your programs to access the Screen.H functions, simply add the following line to the program:

```
#include "Screen.H"
```

In addition, the following three header files are used. These three header files are included automatically by Screen.H, so your programs do not need to specifically include them.

Header File	Contents
PCkeys.H	The definitions for the special keys handled by Inkey( )
PCcolors.H	The definitions for the colors that can be used by DrawBox( ), SayColor( ), and several other functions
PCbox.H	The definitions of the box-drawing characters used in DrawBox( )

THE SOURCE CODE

The source code for the Screen Library is in the following files: Screen.H, PCcolors.H, PCbox.H, PCkeys.H, and Screen.C. Each of these files are listed and discussed below.

THE SCREEN.H HEADER FILE

The header file for the entire library is Screen.H. This file contains the prototypes for all the functions, as well as several #define statements to help the readability of your programs. In addition, a short description of each function is given, creating documentation for the library.

Include Header Files One Time Only

Header files should only be included once in your source programs. To make sure you follow this rule, you can use special **#ifndef** and **#endif** statements at the top and bottom of each header file. When you study Program P8-1, the Screen.H file, you'll notice the following two lines at the top of the file:

```
#ifndef SCREEN_H    /* Make sure this file is only
                       included once.*/
#define SCREEN_H
```

Notice, also, the last line in the Screen.H file:

```
#endif  /* defined SCREEN_H */
```

This set of commands first checks to see if SCREEN_H is defined (SCREEN_H is the name of the header file, with the period replaced by an underscore). If SCREEN_H has not been defined, it gets defined in the next **#define statement**. Then the rest of the header file is processed, down to the last **#endif**. Any further **#includes** of this file will skip the code between the **#ifndef** and **#endif**, because SCREEN_H has already been defined.

Header files are explained in depth in Chapter 11.

Program P8-1, which follows, is a listing of the Screen.H file.

P8-1

```
/*-------------------------------------------------------------------*
 *  File:         P8-1.H    (a.k.a. Screen.H)
 *                  *
    Header for Screen.C
 *  Author:       L. John Ribar
 *                C DiskTutor
 *  Date:         28 December 1992
 *
 *-------------------------------------------------------------------*/

#ifndef SCREEN_H   /* Make sure this file is only included
                      once.*/
#define SCREEN_H
```

```
#include "PCbox.H"        /* Includes the needed header files,
                          /* so you won't forget one
#include "PCcolors.H"        accidentally */

#include "PCkeys.H"

/**************************************************************/
/**                                                         **/
/**                SCREEN MANAGER FUNCTIONS                 **/
/**                                                         **/
/**************************************************************/


/*----------------------------------------------------------*/

#define NORMAL 7       /* Normal attribute, white on black */
#define CLEAR  0       /* Number of rows to scroll for Clearing
                          area */


/*---------------------------------------------------------
   Move() moves the cursor to a specified screen location.
  ---------------------------------------------------------*/
void Move( unsigned row, unsigned col );


/*---------------------------------------------------------
   Where() returns the current cursor location. For ANSI,
   this returns -1,-1, because of the complexity of finding
   the information through ANSI sequences.
  ---------------------------------------------------------*/
void Where( unsigned *row, unsigned *col );


/*---------------------------------------------------------
   Say() writes the given text at the row and column
   specified, using the current attributes.
  ---------------------------------------------------------*/
void Say( unsigned row, unsigned col, char *text );


/*---------------------------------------------------------
   Attr() is a utility function that creates an attribute
   byte from separate foreground and background colors.
  ---------------------------------------------------------*/
```

```
char Attr( int fore, int back );


/*------------------------------------------------------------
   ColorsOf() is a utility function that breaks an
   attribute byte into separate foreground and background
   colors.
  ------------------------------------------------------------*/
void ColorsOf( char attr, int *fore, int *back );


/*------------------------------------------------------------
   SayColor() is used to write text at a given position,
   using specified foreground and background colors.
  ------------------------------------------------------------*/
void SayColor( unsigned row, unsigned col, char *text, char
               attr );


/*------------------------------------------------------------
   Inkey() reads the keyboard and returns the character
   the user typed. For special characters, the file
   PCkeys.H contains the values that will be returned.
  ------------------------------------------------------------*/
char Inkey( void );


/*------------------------------------------------------------
   Keypressed() returns 1 (True) if a key has been pressed
   by the user, without reading the key. If no key has
   been pressed, a 0 (False) is returned.
  ------------------------------------------------------------*/
int Keypressed( void );


/*------------------------------------------------------------
   clrscr() clears the screen.
  ------------------------------------------------------------*/
void clrscr( void );


/*------------------------------------------------------------
   Shadow() is used to place a shadow to the lower-right of
   the area specified.
  ------------------------------------------------------------*/
```

```
void Shadow( int row1, int col1, int row2, int col2,
             char attr );


/*----------------------------------------------------
   Scroll() is used to scroll an area of the screen. The
   empty row created is colored with the supplied attribute.
   If moves is positive, the scroll is downward; if moves
   is negative, the scroll is upward; if zero, the area is
   cleared.
  ----------------------------------------------------*/
void Scroll( unsigned row1, unsigned col1, unsigned row2,
             unsigned col2, int moves, char Att );


/**************************************************************/
/**                                                          **/
/**        TEXT-BASED USER INTERFACE (TUI) FUNCTIONS         **/
/**                                                          **/
/**************************************************************/


/*----------------------------------------------------
   ResetArea() changes the character and attribute of a
   specified area of the screen to a given attribute.
  ----------------------------------------------------*/
void ResetArea( int r1, int c1, int r2, int c2, char ch, char
                attr);


/*----------------------------------------------------
   Fill character definitions for Backdrop()
  ----------------------------------------------------*/
#define LIGHT_FILL   0xb0
#define MEDIUM_FILL  0xb1
#define DARK_FILL    0xb2


/*----------------------------------------------------
   Backdrop() is used to clear the screen to a specific
   character and attribute. This serves as a background
   for the TUI functions.

   Once Backdrop() is called, the character and colors
   used are saved, so that ReplaceBack() can reset
```

```
   portions of the screen to the background look. The
   character and colors are saved in 2 static variables.
  -------------------------------------------------------------*/
void Backdrop( char ch, char attr );


/*-------------------------------------------------------------
   ReplaceBack() covers a portion of the screen with the
   saved background character and attribute.
  -------------------------------------------------------------*/
void ReplaceBack( int row1, int col1, int row2, int col2 );


/*-------------------------------------------------------------
   Definitions for use with DrawBox() to aid understanding
  -------------------------------------------------------------*/

#define FILL 1        /* whether to fill the box or not */
#define NO_FILL 0
#define SHADOW 1      /* whether to draw a shadow on box */
#define NO_SHADOW 0


/*-------------------------------------------------------------
   DrawBox() draws a box on the screen, using the charac-
   ters and attributes specified. If IBM line characters
   are given, as defined in box.h, the corners are made
   to match. If fill is true, the area within the box is
   filled with the same attribute. If shadow is true, a
   shadow is added to the box.
  -------------------------------------------------------------*/
void DrawBox( int row1, int col1, int row2, int col2,
              int vert, int horiz, char attr, int fill, int
              shadow );


/*-------------------------------------------------------------
   HELP Line processing routines
  -------------------------------------------------------------*/

#define NO_HELP (-1)    /* row = NO_HELP disables help */


/*-------------------------------------------------------------
   HelpSetup() sets the row and color of the help messages.
```

```
   If the row is specified as NO_HELP, then all SayHelp()
   calls are ignored. Defaults:  row 24 and colors Blue
   on Gray.
  -----------------------------------------------------------*/
void HelpSetup( unsigned row, char attr );


/*----------------------------------------------------------
   ClearHelp() clears the Help line, if help is enabled.
  -----------------------------------------------------------*/
void ClearHelp( void );        /* clears help message */


/*----------------------------------------------------------
   SayHelp() displays the given message on the help line,
   if help is enabled. Also, clears previous contents.
  -----------------------------------------------------------*/
void SayHelp( char *txt );


/*----------------------------------------------------------
   TITLE Line processing routines
  -----------------------------------------------------------*/

#define NO_TITLE (-1)    /* row = NO_TITLE disables title */


/*----------------------------------------------------------
   TitleSetup() sets the row and color of help messages.
   If row is specified as NO_TITLE, then all SayTitle()
   calls are ignored. Defaults: row 0 and colors Red on
   Gray.
  -----------------------------------------------------------*/
void TitleSetup( unsigned row, char attr );


/*----------------------------------------------------------
   ClearTitle() clears the Title line, if title is enabled.
   Also places the C DiskTutor banner at the right margin.
  -----------------------------------------------------------*/
void ClearTitle( void );


/*----------------------------------------------------------
   SayTitle() displays the given message on the title line,
```

```
   if title is enabled. Also, clears previous contents.
  -----------------------------------------------------------*/
void SayTitle( char *txt );


/*-----------------------------------------------------------
   Message Box processing routines
  -----------------------------------------------------------*/


/*-----------------------------------------------------------
   MessageBox() places a shadowed box in the center of the
   screen, large enough to hold the specified message.
  -----------------------------------------------------------*/
void MessageBox( char *text );


/*-----------------------------------------------------------
   ClearMessageBox() clears previous MessageBox presented,
   replacing the backdrop under the box.
  -----------------------------------------------------------*/
void ClearMessageBox( void );


#endif  /* defined SCREEN_H */
```

THE PCCOLORS.H HEADER FILE

The header file called PCcolors.H includes definitions for the colors available on a standard PC color monitor. These are used to aid the readability of your programs by using color names rather than numbers. Notice that each color name begins with a small *c* (for *color*), followed by the name of the color. The numbers assigned to the color names are those used by the BIOS screen-handling routines to set colors on a PC screen.

Program P8-2, which follows, shows the PCcolors.H file.

P8-2

```
/*-----------------------------------------------------------*
 *  File:        P8-2.H    (a.k.a. PCcolors.H)
 *               *
 *  Header for screen colors
 *  Author:      L. John Ribar
 *               C DiskTutor
```

```
 *  Date:        28 December 1992
 *
 *-----------------------------------------------------------*/

#ifndef PCCOLORS_H
#define PCCOLORS_H

#define cBlack          0
#define cBlue           1
#define cGreen          2
#define cCyan           3
#define cRed            4
#define cMagenta        5
#define cBrown          6
#define cGray           7

#define cDarkGray       8
#define cLightBlue      9
#define cLightGreen    10
#define cLightCyan     11
#define cLightRed      12
#define cLightMagenta 13
#define cYellow        14
#define cWhite         15

/* Add cBlink to a background to cause blinking */
#define cBlink 8

#endif  /* defined PCCOLORS_H */
```

THE PCBOX.H HEADER FILE

The header file called PCbox.H contains definitions for the IBM box-drawing characters used by the **DrawBox()** function. These definitions include characters for single- and double-walled boxes. Four types of boxes can be drawn, and **DrawBox()** will match the corners correctly (see Figure 8-2).

Program P8-3, which follows, shows the PCbox.H file. The numbers specified for each definition are the ASCII character codes required to draw the figure (see Appendix B).

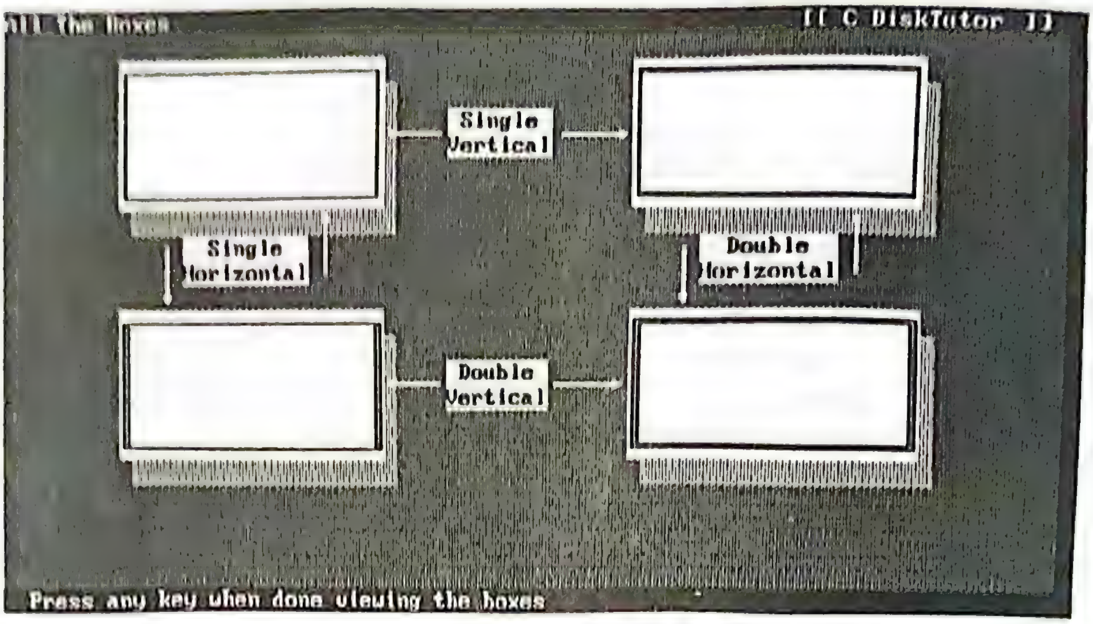


FIGURE 8-2
Four types of boxes that DrawBox( ) *can draw*

P8-3

```
/*--------------------------------------------------------------*
 *  File:       P8-3.H      (a.k.a. PCbox.H)
 *              *
    Header for box-drawing characters
 *  Author:     L. John Ribar
 *              C DiskTutor
 *  Date:       28 December 1992
 *
 *--------------------------------------------------------------*/

#ifndef PCBOX_H
#define PCBOX_H


/* Definitions for IBM box-drawing characters */

#define HORIZ   (0xc4)  /* single horizontal line */
#define D_HORIZ (0xcd)  /* double horizontal line */
#define VERT    (0xb3)  /* single vertical line   */
#define D_VERT  (0xba)  /* double vertical line   */
```

```
/* Definitions for corners, depending on HORIZ and VERT */

#define UL (0xda)
#define UR (0xbf)      /* HORIZ and VERT */
#define LL (0xc0)
#define LR (0xd9)

#define D_UL (0xc9)
#define D_UR (0xbb)    /* D_HORIZ and D_VERT */
#define D_LL (0xc8)
#define D_LR (0xbc)

#define HD_UL (0xd5)
#define HD_UR (0xb8) /* D_HORIZ and VERT */
#define HD_LL (0xd4)
#define HD_LR (0xbe)

#define VD_UL (0xd6)
#define VD_UR (0xb7) /* HORIZ and D_VERT */
#define VD_LL (0xd3)
#define VD_LR (0xbd)

#endif  /* defined PCBOX_H */
```

THE PCKEYS.H HEADER FILE

The header file called PCkeys.H contains definitions for nearly all the special characters available on the IBM PC. These include

- The function keys (F1 through F12; SHIFT-F1 through SHIFT-F12; CTRL-F1 through CTRL-F12; and ALT-F1 through ALT-F12)
- The arrow and other cursor-movement keys (UP ARROW, DOWN ARROW, LEFT ARROW, RIGHT ARROW, PGUP, PGDN, HOME, and END)
- The ALT-key versions of the alphabetic characters (ALT-A through ALT-Z)
- Miscellaneous other keys

Program P8-4, which follows, shows the PCkeys.H header file. To derive these key numbers, the keyboard scan code for these keys is OR'd with the value 0x80. This results in key values that do not conflict with the standard keyboard characters. (For a more complete discussion of keyboard scan codes and keyboard handling routines, refer to the PC programming

books listed in the Bibliography.) Also, notice that the value for each definition is listed as an octal character (a leading backslash followed by three octal digits), as is done in the format portion of **printf( )** statements.

P8-4

```
/*-----------------------------------------------------------*
 *  File:        P8.4.H      (a.k.a. PCkeys.H)
 *               *
 *  Header for special keystrokes
 *  Author:      L. John Ribar
 *               C DiskTutor
 *  Date:        28 December 1992
 *
 *----------------------------------------------------------*/
#ifndef PCKEYS_H
#define PCKEYS_H

/*
   Function and special key definitions for use as constants
   with InKey()
*/

#define CR     '\015'
#define Esc    '\033'
#define BackSp '\010'
#define Tab    '\011'

#define ShTab  '\217'
#define SysReq '\362'

#define F1     '\273'
#define F2     '\274'
#define F3     '\275'
#define F4     '\276'
#define F5     '\277'
#define F6     '\300'
#define F7     '\301'
#define F8     '\302'
#define F9     '\303'
#define F10    '\304'
#define F11    '\205'
#define F12    '\206'

#define LtArrow '\313'
#define RtArrow '\315'
#define UpArrow '\310'
```

```
#define DnArrow '\320'

#define PGDN    '\321'
#define PGUP    '\311'
#define HOMEKey '\307'
#define ENDKey  '\317'
#define INSKey  '\322'
#define DELKey  '\323'
#define CtlPgUp '\204'
#define CtlPgDn '\366'
#define CtlEnd  '\365'
#define CtlHome '\367'
#define CtlLt   '\363'
#define CtlRt   '\364'

#define ShF1   '\324'
#define ShF2   '\325'
#define ShF3   '\326'
#define ShF4   '\327'
#define ShF5   '\330'
#define ShF6   '\331'
#define ShF7   '\332'
#define ShF8   '\333'
#define ShF9   '\334'
#define ShF10  '\335'
#define ShF11  '\207'
#define ShF12  '\210'

#define CtlF1  '\336'
#define CtlF2  '\337'
#define CtlF3  '\340'
#define CtlF4  '\341'
#define CtlF5  '\342'
#define CtlF6  '\343'
#define CtlF7  '\344'
#define CtlF8  '\345'
#define CtlF9  '\346'
#define CtlF10 '\347'
#define CtlF11 '\211'
#define CtlF12 '\212'

#define AltF1  '\350'
#define AltF2  '\351'
#define AltF3  '\352'
#define AltF4  '\353'
```

```
#define AltF5  '\354'
#define AltF6  '\355'
#define AltF7  '\356'
#define AltF8  '\357'
#define AltF9  '\360'
#define AltF10 '\361'
#define AltF11 '\213'
#define AltF12 '\214'

#define Alt1   '\370'
#define Alt2   '\371'
#define Alt3   '\372'
#define Alt4   '\373'
#define Alt5   '\374'
#define Alt6   '\375'
#define Alt7   '\376'
#define Alt8   '\377'
#define Alt9   '\200'
#define Alt0   '\201'

#define AltA   '\236'
#define AltB   '\260'
#define AltC   '\256'
#define AltD   '\240'
#define AltE   '\222'
#define AltF   '\241'
#define AltG   '\242'
#define AltH   '\243'
#define AltI   '\227'
#define AltJ   '\244'
#define AltK   '\245'
#define AltL   '\246'
#define AltM   '\262'
#define AltN   '\261'
#define AltO   '\230'
#define AltP   '\231'
#define AltQ   '\220'
#define AltR   '\223'
#define AltS   '\237'
#define AltT   '\224'
#define AltU   '\226'
#define AltV   '\257'
#define AltW   '\221'
#define AltX   '\255'
```

```
#define AltY   '\225'
#define AltZ   '\254'

#endif   /* defined PCKEYS_H */
```

THE SOURCE FILE

The file called Screen.C contains the actual code that implements the Screen and TUI Library. Before you start reading the sections that follow, bring the Screen.C file into the DiskTutor Environment so you can view it along with the discussion here.

ANSI and BIOS Support

In this section you'll get a technical overview of the ANSI and BIOS interfaces; though it gives you general information on these concepts, the details are beyond the scope of this book.

One of the requirements for this library is support for both BIOS and ANSI screen-access methods. The ANSI interface is implemented through sequences of characters that ANSI drivers understand (see Appendix B). The BIOS interface is implemented using interrupt calls to the PC BIOS (Basic Input and Output System) that is built into all IBM-compatible personal computers.

The ANSI version of the Screen Library requires that you have an ANSI driver loaded on your PC, or that you are using an ANSI-compatible display. Without this file, the ANSI version of the Screen Library will not work, and your programs will display unreadable screens. To load the ANSI screen driver, enter the following line in your system's CONFIG.SYS file (this command assumes that the file ANSI.SYS exists in your root (\) directory):

```
device=ansi.sys
```

To keep both versions of the library in one file, preprocessor **#define** and #ifdef statements are used (these are fully explained in Chapter 11).

At the top of the Screen.C file, you must place one or the other of the following statements:

```
#define screenANSI
/*      OR        */
#define screenBIOS
```

Caution: Use only one of these statements, not both, or you will get inconsistent results. Remember that some of the library functions, notably **Scroll( )** and **Keypressed( )**, cannot be implemented using ANSI sequences (see Appendix B for more information about the ANSI sequences).

Now observe that most of the screen-handling functions have an #ifdef statement, as shown here for **Move( )**:

```
/*-------------------------------------------------------------
   Move() moves the cursor to a specified screen location.
  -------------------------------------------------------------*/
void Move( unsigned row, unsigned col )
{

#ifdef screenANSI

   char command[10];
   sprintf( command, “%d;%dH”, row, col );
   SayANSI( command );

#else /* BIOS */

   REGISTERS regs;

   AH = 2;    /* interrupt function */
   BH = 0;    /* page number */
   DH = row;
   DL = col;
   int86( 0x10, &regs, &regs );

#endif

}
```

If the **#define screenANSI** statement is placed at the top of the file, the statements between

```
#ifdef screenANSI
```

and

```
#else /* BIOS */
```

are executed. If **screenBIOS** is defined instead, the code between

```
#else /* BIOS */
```

and

```
#endif
```

is executed.

The ANSI sequences are sent to the screen using the **SayANSI()** function. This function prints the specific character sequence, preceded by the escape character, and an open square bracket character, [. This is how all ANSI commands begin, and are thus recognized by the ANSI driver.

The BIOS interface is designed around a set of preprocessor **#define** statements, as shown here; you'll see these included in the Screen.C file, which comprises the next program, Program P8-5.

```
#include <dos.h>

/* Define compiler-specific handling of registers for BIOS
   calls */

#define REGISTERS union REGS

#define AX regs.w.ax
#define AL regs.h.al
#define AH regs.h.ah

#define BX regs.w.bx
#define BL regs.h.bl
#define BH regs.h.bh

#define CX regs.w.cx
#define CL regs.h.cl
#define CH regs.h.ch

#define DX regs.w.dx
#define DL regs.h.dl
#define DH regs.h.dh
```

The dos.h header file included at the top of these **#define** statements contains the prototypes needed to interface your program with the interrupt calls needed in the BIOS interface. The other **#define** statements make

your code both more readable and more portable. If you use another compiler, just change these statements to match how your compiler handles interrupts, and the rest of Screen.C can remain unchanged.

Each of these **#define** statements is equating a symbol, such as AX, to a member of the **regs** union, such as **regs.w.ax**. The **regs** union is defined in dos.h.

Program P8-5, which follows, shows the Screen.C file.

P8-5

```
/*-----------------------------------------------------------*
 *  File:       Screen.C       (a.k.a. P8-5.C)
 *  Author:     L. John Ribar
 *              C DiskTutor
 *  Date:       28 Dec 1992
 *
 *-----------------------------------------------------------*
 *  Revision history:
 *              12/28/92 Initial Coding
 *-----------------------------------------------------------*/

#include <string.h>
#include <conio.h>
#include "Screen.H"

/*
   Define screenANSI or screenBIOS at this point to select
   type of processing you want for screen calls.
*/
#define screenBIOS


/*-----------------------------------------------------------*/

#ifdef screenANSI

/*
   SayANSI prints required ANSI code prelude ( ESC+'[' ),
   and then prints command string passed in parameter 'txt'.
*/
void SayANSI( char *txt )
{
   printf("%c%c%s", 27, 91, txt );
}

#else /* BIOS */
```

```
#include <dos.h>

/* Define compiler-specific handling of registers for BIOS
   calls */

#define REGISTERS union REGS

#define AX regs.w.ax
#define AL regs.h.al
#define AH regs.h.ah

#define BX regs.w.bx
#define BL regs.h.bl
#define BH regs.h.bh

#define CX regs.w.cx
#define CL regs.h.cl
#define CH regs.h.ch

#define DX regs.w.dx
#define DL regs.h.dl
#define DH regs.h.dh

#endif


/*-----------------------------------------------------------
   Move() moves the cursor to a specified screen location.
  -----------------------------------------------------------*/
void Move( unsigned row, unsigned col )
{

#ifdef screenANSI

   char command[10];
   sprintf( command, "%d;%dH", row, col );
   SayANSI( command );

#else /* BIOS */

   REGISTERS regs;

   AH = 2;    /* interrupt function */
   BH = 0;    /* page number */
```

```
   DH = row;
   DL = col;
   int86( 0x10, &regs, &regs );

#endif

}


/*-------------------------------------------------------------
   Where() returns the current cursor location. For ANSI,
   this returns -1,-1, because of complexity of finding
   the information through ANSI sequences.
  -------------------------------------------------------------*/
void Where( unsigned *row, unsigned *col )
{

#ifdef screenANSI

   *row = -1;
   *col = -1;

#else /* BIOS */

   REGISTERS regs;
   AH = 3;    /* interrupt function */
   BH = 0;    /* page number */
   int86( 0x10, &regs, &regs );
   *row = DH;     /* high byte of D */
   *col = DL;     /* low byte of D */

#endif

}


/*-------------------------------------------------------------
   Say() writes the given text at the row and column
   specified, using the current attributes.
  -------------------------------------------------------------*/
void Say( unsigned row, unsigned col, char *text )
{

#ifdef screenANSI
```

```
   Move( row, col );
   puts( text );

#else /* BIOS */

   REGISTERS regs;

   Move( row, col );

   while (*text)
   {
      AH = 0x0A;    /* interrupt function */
      BH = 0;       /* page number*/
      AL = *text;
      CX = 1;          /* number of times to write the
                          character */
      int86( 0x10, &regs, &regs );
      text++;          /* next position */
      Move( row, ++col );
   }

#endif

}


/*-----------------------------------------------------------
   Attr() is a utility function that creates an attribute
   byte from separate foreground and background colors.
  -----------------------------------------------------------*/
char Attr( int fore, int back )
{
   return (char) ((back << 4)|fore);
}


/*-----------------------------------------------------------
   ColorsOf() is a utility function that breaks an attri-
   bute byte into separate foreground and background colors.
  -----------------------------------------------------------*/
void ColorsOf( char attr, int *fore, int *back )
{
   *fore = attr & 15;
   *back = (attr >> 4) & 15;
```

```
}


/*----------------------------------------------------------
   SayColor() is used to write text at a given position,
   using specified foreground and background colors.
  ----------------------------------------------------------*/
void SayColor( unsigned row, unsigned col, char *text, char
               attr )
{

#ifdef screenANSI

   int cvtTable[] =
    /* BLA BLU GRE CYA RED MAG YEL WHI */
      { 30, 34, 32, 36, 31, 35, 33, 37,  /* foregrounds */
        40, 44, 42, 46, 41, 45, 43, 47 };
   int fore, back;
   char command[15];

   ColorsOf( attr, &fore, &back );
   sprintf( command, "0;%d;%d", cvtTable[fore % 8],
            cvtTable[(back % 8) +8] );
   if (fore >= 8)
      strcat( command, ";1" );   /* Add high intensity */
   if (back >= 8)
      strcat( command, ";5" );   /* Add blinking */
   strcat( command, "m" );       /* ANSI attribute command =
                                    'm' */
   SayANSI( command );           /* Change the color */
   Say( row, col, text );        /* Print the text */

#else  /* BIOS */

   REGISTERS regs;

   Move( row, col );

   while (*text)
   {
      AH = 9;      /* interrupt function */
      BH = 0;      /* page number */
      AL = *text;
      BL = attr;
      CX = 1;        /* number of times to write the
```

```
                          character */
      int86( 0x10, &regs, &regs );
      text++;          /* increment pointer for the next
                          position */
      Move( row, ++col );
   }

#endif

}


/*-------------------------------------------------------------
   Inkey() reads the keyboard, and returns the character
   that the user typed. For special characters, the file
   PCkeys.H contains the values that will be returned.
  -------------------------------------------------------------*/
char Inkey()
{

#ifdef screenANSI

   char ch;

   /* ANSI does not define keyboard input routines, so just
      use library function getch() here. */
   ch = getch();
   if (ch == 0)    /* Extended keys return 0 as the first
                      character, */
   {               /* so read a second character immediately. */
      ch = getch();
      ch |= 0x80;
   }
   return ch;

#else

   REGISTERS regs;

   AH = 0;        /* interrupt function */
   int86( 0x16, &regs, &regs );

   if (AL == 0)
      return (char) AH | 0x80;
   else
```

```
      return (char) AL;

#endif

}


/*-------------------------------------------------------------
   Keypressed() returns 1 (True) if a key was pressed by
   the user, without reading the key. If no key was
   pressed, a 0 (False) is returned.
  -------------------------------------------------------------*/
int Keypressed()
{

#ifdef screenANSI

   return kbhit();

#else

   REGISTERS regs;

   AH = 0x0B;
   int86( 0x21, &regs, &regs );
   if ( AL == 0x0FF )
      return 1;
   else
      return 0;

#endif

}


/*-------------------------------------------------------------
   clrscr() clears the screen.
  -------------------------------------------------------------*/
void clrscr()
{

#ifdef screenANSI

   SayANSI("2J");
```

```
#else

   Scroll( 0, 0, 24, 79, 0, NORMAL );

#endif

}


/*------------------------------------------------------
   ResetArea() changes the character and attribute of a
   specified area of the screen to a given attribute.
  ------------------------------------------------------*/
void ResetArea( int r1, int c1, int r2, int c2, char ch, char
attr)
{

#ifdef screenANSI

   char text[80];
   int i;

   /* First set up the text to print.*/
   for (i=0; i<(c2-c1+1); i++)
      text[i] = ch;
   text[c2-c1+1] = 0;    /* End the string correctly.*/

   /* Now display it for each row needed.*/
   for (i=r1; i<=r2; i++)
      SayColor( i, c1, text, attr );

#else /* BIOS */

   REGISTERS regs;
   int row;

   for (row=r1; row<=r2; row++)
   {
      Move( row, c1 );
      AH = 9;
      BH = 0;
      CX = c2-c1+1;
      AL = ch;
      BL = attr;
```

```
      int86( 0x10, &regs, &regs );
   }

#endif

}


/*------------------------------------------------------------
   Backdrop() is used to clear the screen to a specific
   character and attribute. This serves as a background
   for the text-based user interface (TUI) functions.

   Once this function is called, the character and colors
   used are saved, so that ReplaceBack() can reset
   portions of the screen back to the background look. The
   character and colors are saved in 2 static variables,
   which appear directly below.
  ------------------------------------------------------------*/
static char BackdropChar;
static char BackdropAttr;

void Backdrop( char ch, char attr )
{
   BackdropChar = ch;
   BackdropAttr = attr;
#ifdef screenANSI
   ResetArea( 1, 1, 25, 79, ch, attr );
#else
   ResetArea( 0, 0, 24, 79, ch, attr );
#endif
}


/*------------------------------------------------------------
   ReplaceBack() covers a portion of the screen with the
   saved background character and attribute.
  ------------------------------------------------------------*/
void ReplaceBack( int row1, int col1, int row2, int col2 )
{
   ResetArea( row1, col1, row2, col2, BackdropChar,
              BackdropAttr );
}
```

```
/*-----------------------------------------------------------
   Shadow() is used to place a shadow to the lower-right of
   the area specified.
-----------------------------------------------------------*/
void Shadow( int row1, int col1, int row2, int col2,
             char attr )
{

#ifdef screenANSI

   ResetArea( row1+1, col2+1, row2+1, col2+1, LIGHT_FILL,
              attr ); /* side */
   ResetArea( row2+1, col1+1, row2+1, col2+1, LIGHT_FILL,
              attr ); /* bottom */

#else /* BIOS */

   REGISTERS regs;
   int i, row, col;
   char ch;

   /* First do bottom row. */
   row = row2+1;
   for (col=col1+1; col<=col2+1; col++)
   {
      Move( row, col );
      /* Read current character at the location. */
      AH = 8;     /* interrupt function */
      BH = 0;     /* page number */
      int86( 0x10, &regs, &regs );
      ch = AL;

      /* Now rewrite it with new attribute */
      AH = 9;      /* interrupt function */
      BH = 0;      /* page number */
      AL = ch;
      BL = attr;
      CX = 1;        /* number of times to write the
                        character */
      int86( 0x10, &regs, &regs );
   }

   /* Now do the right side. */
   col = col2+1;
   for (row=row1+1; row<=row2+1; row++)
```

```
   {
      Move( row, col );
      /* Read current character at the location */
      AH = 8;      /* interrupt function */
      BH = 0;      /* page number */
      int86( 0x10, &regs, &regs );
      ch = AL;

      /* Now rewrite it with new attribute. */
      AH = 9;       /* interrupt function */
      BH = 0;       /* page number */
      AL = ch;
      BL = attr;
      CX = 1;         /* number of times to write the
                         character */
      int86( 0x10, &regs, &regs );
   }

#endif

}


/*----------------------------------------------------------
   Scroll() is used to scroll an area of the screen. The
   empty row created is colored with the supplied attribute.
   If moves is positive, the scroll is downward; if moves
   is negative, the scroll is upward; if zero, the area is
   cleared.
  ----------------------------------------------------------
void Scroll( unsigned row1, unsigned col1, unsigned row2,
             unsigned col2, int moves, char Att )
{

#ifdef screenANSI

   return;

#else /* BIOS */

   REGISTERS regs;

   if (moves > 0)
      AH = 7;
   else
```

```
    {
      AH = 6;
      moves *= -1;
    }
    AL = moves;
    CH = row1;
    CL = col1;
    DH = row2;
    DL = col2;
    BH = Att;
    int86( 0x10, &regs, &regs );

#endif

}


/*------------------------------------------------------------
    DrawBox() draws a box on screen, using the characters
    and attributes specified. If IBM line characters are
    given, as defined in box.h, then the corners are made
    to match. If fill is true, the area within the box is
    filled with the same attribute. If shadow is true, a
    shadow is added to the box.
 -----------------------------------------------------------*/
void DrawBox( int row1, int col1, int row2, int col2,
              int vert, int horiz, char attr, int fill, int
              shadow )
{
    int     i, nrows;
    int     ul, ur, ll, lr;
    char    tmp[2];

    if( !( (horiz == HORIZ || horiz == D_HORIZ) &&
           (vert  == VERT  || vert  == D_VERT ) ))
    {
            ul = ur = ll = lr = vert ;
    }
    else
    {
        if( vert == VERT )
        {
            if(horiz==HORIZ)
            {
                ul=UL;    ur=UR;    ll=LL;    lr=LR;
```

```
          }
          else
          {
               ul=HD_UL; ur=HD_UR; ll=HD_LL; lr=HD_LR;
          }
     }
     else
     {
          if(horiz==HORIZ)
          {
               ul=VD_UL; ur=VD_UR; ll=VD_LL; lr=VD_LR;
          }
          else
          {
               ul=D_UL;  ur=D_UR;  ll=D_LL;  lr=D_LR;
          }
     }
}

if (fill)         /* Fill the box with the same color. */
  ResetArea( row1, col1, row2, col2, ' ', attr );
if (shadow)       /* Shadow in default color */
  Shadow( row1, col1, row2, col2, Attr( cGray, cBlack ) );

tmp[1] = 0;

tmp[0] = ul;                               /* corners */
SayColor( row1, col1, tmp, attr );
tmp[0] = ur;
SayColor( row1, col2, tmp, attr );
tmp[0] = ll;
SayColor( row2, col1, tmp, attr );
tmp[0] = lr;
SayColor( row2, col2, tmp, attr );

tmp[0] = horiz;                      /* horizontal lines */
for( i = col1+1; i<col2; i++ )
{
  SayColor( row1, i, tmp, attr );
  SayColor( row2, i, tmp, attr );
}

tmp[0] = vert;                             /* vertical lines */
for( i = row1+1; i<row2; i++ )
{
```

```
        SayColor( i, col1, tmp, attr );
        SayColor( i, col2, tmp, attr );
      }

}


/*---------------------------------------------------------*/

static char spaces[81] =    /* Used by clearField to clear
                               field */
"
";

static void clearField( unsigned row, unsigned col, int len,
char attr )
{
   spaces[len] = 0;
   SayColor( row, col, spaces, attr );
   spaces[len] = ' ';
}


/*--------------------------------------------------------
   HelpSetup() sets the row and color of the help messages.
   If the row is specified as NO_HELP, then all SayHelp()
   calls are ignored. Defaults: row 24 and colors Blue
   on Gray.
  --------------------------------------------------------*/

static unsigned HelpRow = 24;
static char HelpAttr = (char) (( cGray << 4 ) + cBlue);

void HelpSetup( unsigned row, char attr )
{
   if (row)
      HelpRow = row;
   if (attr)
      HelpAttr = attr;
}


/*--------------------------------------------------------
   ClearHelp() clears the Help line, if help is enabled.
  --------------------------------------------------------*/
```

```
void ClearHelp( void )
{
   if (HelpRow == -1)
      return;
#ifdef screenANSI
   clearField( HelpRow, 0, 79, HelpAttr );
#else
   clearField( HelpRow, 0, 80, HelpAttr );
#endif
}


/*-------------------------------------------------------------
   SayHelp() displays the given message on the help line,
   if help is enabled. Also, clears previous contents.
  ------------------------------------------------------------*/
void SayHelp( char *txt )
{
   if (HelpRow == -1)
      return;
   ClearHelp();
   SayColor( HelpRow, 3, txt, HelpAttr );
}


/*-------------------------------------------------------------
   TitleSetup() sets the row and color of the help messages.
   If the row is specified as NO_TITLE, then all SayTitle()
   calls are ignored. Defaults: row 0 and colors Red on
   Gray.
  ------------------------------------------------------------*/

static unsigned TitleRow = 0;
static char TitleAttr = (char) (( cGray << 4 ) + cRed);

void TitleSetup( unsigned row, char attr )
{
   if (row)
      TitleRow = row;
   if (attr)
      TitleAttr = attr;
}


/*-------------------------------------------------------------
```

```
   ClearTitle() clears the Title line, if title is enabled.
   Also placed the C DiskTutor banner at the right margin.
  ----------------------------------------------------------*/
void ClearTitle( void )
{
   if (TitleRow == -1)
      return;
#ifdef screenANSI
   clearField( TitleRow, 0, 79, TitleAttr );
#else
   clearField( TitleRow, 0, 80, TitleAttr );
#endif
   Say( TitleRow, 62, "[[ C DiskTutor ]]");
}


/*----------------------------------------------------------
   SayTitle() displays the given message on the title line,
   if title is enabled. Also, clears previous contents.
  ----------------------------------------------------------*/
void SayTitle( char *txt )
{
   if (TitleRow == -1)
      return;
   ClearTitle();
   SayColor( TitleRow, 1, txt, TitleAttr );
}


/*----------------------------------------------------------
   MessageBox() places a shadowed box in the center of the
   screen, large enough to hold the specified message.
  ----------------------------------------------------------*/

static lastMessageSize = 0;
static lastMessageStart = 40;

void MessageBox( char *text )
{
   int l, b;
   l = strlen(text) + 4;
   lastMessageSize = l + 1;  /* Add 1 for shadow */
   b = 40-(l/2);
   lastMessageStart = b;
   DrawBox( 11, b, 15, b+l, VERT, HORIZ, Attr( cYellow,
```

```
cBlack ), FILL, SHADOW );
   Say( 13, b+2, text );
}


/*------------------------------------------------------------
   ClearMessageBox() clears the previous MessageBox
   presented, replacing the backdrop under the box.
  ------------------------------------------------------------*/
void ClearMessageBox( void )
{
   ReplaceBack( 11, lastMessageStart, 16,
lastMessageStart+lastMessageSize );
}


/*------------------------------------------------------------
   The main() test function. Use #define MAIN to enable
   the test code, or remove that #define to remove the
   test code.
/*------------------------------------------------------------*/

#define MAIN

#ifdef MAIN
main()
{
   char ch;          /* Character for keypresses */
   int direction;    /* Direction to scroll; only works
                        with BIOS mode, not with ANSI */

   clrscr();         /* First clear the screen. */

   /* The next three statements create user interface. */
   Backdrop( DARK_FILL, Attr( cWhite, cBlack ) );
   SayTitle( " Screen Library Test Program" );
   SayHelp("Press F1 for Help, F10 to Exit");

   /* Display a message to scroll later. */
   MessageBox( "Hello  There!");
   ch = Inkey();     /* Read a key from the user. */
   direction = 1;    /* Start with scroll going down. */
   while (ch != F10)
   {
```

```
        Scroll( 12, lastMessageStart + 1, 14,
           lastMessageStart + lastMessageSize - 2, direction,
           Attr( cYellow, cBlack ) );  /* Scroll the message. */
        direction *= -1;      /* Reverse the scroll direction. */
        ch = Inkey();                 /* Read another keystroke. */
        if (ch == F1)
           SayHelp("You asked for more help than I have!");
     }
     ClearMessageBox();
     SayHelp("Press any key to quit.");
     ch = Inkey();
}
#endif

/*-------------------------------------------------------*/
```

AN EXAMPLE APPLICATION FOR THE SCREEN LIBRARY

The Screen.C file has a built-in test routine at the end, which you can see in action if you run the program. However, Program P8-6, Boxes.C, uses the Screen Library the way it is meant to be used. To run this program, follow these steps:

1. Edit the Screen.C file and remove the #define MAIN line (look for it just a few lines up from the end of the program).
2. Call the compiler with *both* of the program file names on the command line, like this:

```
wcl boxes screen
```

The compiler will compile both files and link them together, creating an executable program with the name of the first file, in this case BOXES.EXE. If you are in the C DiskTutor environment, use the Compile-Build menu option, and specify both the boxes and screen file names.

Program P8-6, shown next, shows the Boxes.C program, which generated the screen in Figure 8-2. This shows you all the options for boxes that can be drawn, and will give you an idea of how you can use the Screen Library from within your own programs.

P8-6

```
/*
   Program:          Boxes.C     (a.k.a. P8-6.C)
   Purpose:          Show how to use the Screen Library by
                     drawing all four box types.
   Build Sequence: WCL BOXES SCREEN
   By:               L. John Ribar
                     C DiskTutor
   Date:             12/30/92
*/
#include "Screen.H"

main()
{
   char ch;

   /* First, clear the screen. */
   clrscr();

   /* Put up our TUI. */
   Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
   SayTitle( "All The Boxes" );
   SayHelp( "Press any key when done viewing the boxes");

   /* Draw the four boxes. */
   DrawBox( 3, 10, 8, 30,  VERT,  HORIZ, NORMAL, FILL,
            SHADOW );
   DrawBox(13, 10,18, 30, D_VERT,  HORIZ, NORMAL, FILL,
           SHADOW );
   DrawBox( 3, 50, 8, 70,  VERT, D_HORIZ, NORMAL, FILL,
            SHADOW );
   DrawBox(13, 50,18, 70, D_VERT, D_HORIZ, NORMAL, FILL,
           SHADOW );

   /* Now, notate the drawings. */
   Say( 10, 15, "  Single  ");
   Say( 11, 15, "Horizontal");
   Say( 10, 55, "  Double  ");
   Say( 11, 55, "Horizontal");
   Say(  5, 36, " Single ");
   Say(  6, 36, "Vertical");
   Say( 15, 36, " Double ");
   Say( 16, 36, "Vertical");

   /* Leave the boxes displayed until a key is pressed. */
   ch = Inkey();
```

```
    /* Clean up the screen. */
    clrscr();
}
```

Case Study 1: A Programmer's Calculator

This case study will give you more experience using the C programming concepts that you've learned so far. The program you develop in this case study is a programmer's calculator. You'll want to name the program file appropriately, for example, Calc.C. Try to complete the project outlined here on your own, but if you need help or ideas, you can consult the solutions presented in Chapter 13.

Design and build a program that can do the following:

- Run from the command line, or run interactively. This means that you want to allow the user to enter commands on the command line, in which case the program just prints an answer. Alternatively, if no parameters are entered on the command line, the program should interactively ask for commands and numbers from the user, and print results as they are calculated.
- Accept commands or numbers without needing any mode changing or special handling. This requires an input function that can read multiple words from the user at one time (similar to **scan.c**, developed in Chapter 7).
- Print the answer in decimal, octal, and hexadecimal notations.

Here are some optional tasks to try including in your program:

- Use the Screen (TUI) Library developed in this chapter to make the program look more professional. Try making the screen look like the tape on a manual calculator, scrolling the user's entries as they are processed.

Remember: If you want to use the Screen Library functions, you must include the Screen.H header file in your program, and add **screen** to the compiler command line, like this:

```
wcl calc screen
```

Case Study 1: A Programmer's Calculator (*continued*)

This command will compile the Calc.C and Screen.C files, and then link them together to create the executable file calc.exe.

- Add memory functions (such as place in memory, recall from memory, add to memory, and so forth) to the calculator. Try a single-memory calculator, and then multiple memories.
- Allow the numbers and commands to be shown on the screen, and at the same time printed at the printer as a hardcopy reference. (You may need to read Chapter 9, and then come back and add this function to the calculator.)

Use your imagination, and write the calculator program in a way that will be useful to *you*! Remember to follow the design steps outlined in Chapter 2; determine everything that you want the program to do before writing any code. Once the program is completed, be sure to write a one- or two-page document describing how to use it.

One way of completing this project is presented in Chapter 13.

C H A P T E R 9

INPUT AND OUTPUT

Throughout this book, you have seen examples of the C input and output facilities, mostly through the use of the **printf()** and **scanf()** routines. Here in Chapter 9 you'll explore C input and output in more depth. You'll learn about managing input and output from and to the user, strings, and disk files.

> **Note:** The C language itself does not include any functions for input and output. However, the ANSI C definition specifies a default set of library routines for input and output; these are discussed in this chapter.

THE STDIO.H AND HEADER FILES

In most of the example programs in this book you'll see the following line of code:

```
#include <stdio.h>
```

This line is a *precompiler directive* that tells the compiler to read the file stdio.h at that point in the compilation. Why read this file?

Though header files and preprocessor directives are covered more fully in Chapter 11, the special stdio.h file bears special mention here. This header file contains definitions, structures, and prototypes for the standard input and output (I/O) functions of C. These functions are classified in three types: user I/O, string I/O, and file I/O. In this chapter you will study the functions used to manage each of these I/O types.

Tip: As you work through this chapter, notice that the functions for user I/O, string I/O, and file I/O are written quite similarly. The differences involve the spelling of the functions, for example, printf( ), sprintf( ), and fprintf( ); and the source or destination of the data (user, string, or file).

INPUT AND OUTPUT FROM AND TO THE USER

Working with the user involves getting input from the user via the keyboard, and sending output back to the user via the monitor. Several commands allow you to perform these functions.

- **getc( )** and **putc( )** work with single bytes, or characters, as do **getchar( )** and **putchar( )**.
- **gets( )** and **puts( )** deal with continuous strings of characters.
- **scanf( )** and **printf( )**, which you have seen before, handle formatted data.

SINGLE-CHARACTER I/O: GETCHAR( ) AND PUTCHAR( )

The **getchar( )** and **putchar( )** functions are the most basic of user I/O functions. Their prototypes are as follows:

```
#include <stdio.h>

int getchar( void );
int putchar( int c );
```

The **getchar()** function returns an integer value, equivalent to the ASCII code of the keyboard character pressed by the user. This function retrieves input from the keyboard (known as the *standard input device* or stdin), one keystroke at a time. I/O is buffered under DOS, however, so each keystroke is not actually available until the carriage return character occurs. (*Buffered data* is data held by the operating system until a specific character, such as the carriage return, is detected.)

The **putchar()** function performs the opposite function—it moves a single character out to the screen, known as the standard output device, or **stdout**.

Program P9-1 is a short program that reads characters input by the user and then echoes them back to the screen. Load this file into the DiskTutor Environment now.

P9-1

```
/*
   C program to read and write user input.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
    int ch;    /* Character used for input and output */

    do
    {
        ch = getchar();     /* Read one character. */
        putchar(ch);        /* Output the character. */
    } while (ch != 'Q');    /* Continue until Q is pressed. */
}
```

As mentioned just above, input under DOS is buffered, and Program P9-1 shows you what that means. Although each key is read individually, the keystrokes are not passed to the program until the carriage return is pressed. Thus this program makes it look like **getchar()** and **putchar()** work on entire lines, but in reality, that illusion is actually a function of DOS holding keystrokes until a carriage return is pressed, as shown in the following program's output.

These lines were typed by the user

```
C:\CDT\CHAP9>p9-1
This is a test.
This is a test.
It looks like the line is read all at once!
It looks like the line is read all at once!
Q
Q
C:\CDT\CHAP9>
```

These lines were printed by the program

Notice, also, that both the **getchar( )** and **putchar( )** functions utilize integer variables, not character variables. Actually, the work is done internally with unsigned characters, but in C the data is moved around between functions as integers.

STRING I/O: GETS( ) AND PUTS( )

Whereas **getchar( )** and **putchar( )** work with single characters, **gets( )** and **puts( )** work with groups of characters, such as words in a book. The **gets( )** and **puts( )** functions have the following prototypes:

```
#include <stdio.h>

char *gets( char *s );
int puts( const char *s );
```

The **gets( )** function reads a string from the user. The parameter passed into the function is filled with the keystrokes the user presses, and the function returns a pointer to the string just read. You should always check the return value; a NULL pointer is returned if no input was received or if an error occurred. Notice that **gets( )** does not leave a carriage return, pressed by the user to end the input, at the end of the input string.

The companion function to **gets( )** is **puts( )**, which adds a carriage return to the end of each string it displays. The following example, Program P9-2, reads words from the user and prints them out using **gets( )** and **puts( )**. Load and run this program in the DiskTutor Environment, to see how the strings are handled.

P9-2

```
/*
   C program to read and write words with the user.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
     char word[80];  /* Long enough for a full line */
     char *w;

     do
     {
       puts("Enter a word (Q to quit): ");
       w = gets(word);
       if (w == NULL)
           puts("Error occurred during input!");
       else
       {
          puts(w);    /* Put string out using pointer. */
          puts(word); /* Put string out using variable. */
       }
     } while (word[0] != 'Q');  /* Q for Quit */
     puts("Thanks!");
}
```

Using **gets()** is quite straightforward: A character-string pointer is passed in as the only parameter, and the input from the user is copied to that string. The pointer is also returned by the function. Therefore, as in Program P9-1, the string you enter is copied to **word**, and the address of the string is returned as **w**. As explained previously, a NULL return value means there was no input, or an error occurred.

Caution: When using **gets()**, keep in mind the possibility that the user may enter too many characters. The pointer passed to **gets()** must point to enough space to handle the worst case, or your program may crash.

FORMATTED I/O: SCANF( ) AND PRINTF( )

The **scanf( )** and **printf( )** are probably the most versatile and easiest to use of C's routines. These helpful, basic functions let you read or print one or more items, in a format you specify. The prototypes for these routines are as follows:

```
#include <stdio.h>

int printf( const char *format, ... );
int scanf( const char *format, ... );
```

About Format Specifiers The ellipsis (...) at the end of both these calls means that a variable number of arguments may follow the required *format* parameter. The *format* parameter may have nearly anything in it, including special formatting commands or *specifiers*. Let's take a look at these specifiers; they all begin with the percent character (%), and have this general form:

```
%[flags] [width] [.precision] [modifier] type
```

where the parameters enclosed in square brackets are optional. Following is an explanation of each of the parameters used with specifiers.

The *flags* parameter describes special commands that will apply to the item when it is printed. These flags are acceptable:

Flag Character	Purpose
-	Left-justifies a field (fields are normally right-justified)
+	Forces a plus or minus sign to always appear in the field (normally, a minus sign shows for negative numbers, but nothing is displayed for positive numbers)
Space	Generates a space for numbers that have neither a plus nor a minus sign
#	Puts a 0 at the beginning of an octal number, or 0X in front of a hexadecimal number
0	Pads the start of the number with 0's
*	With **scanf( )**, ignores the field that is read

The *width* parameter specifies the minimum size (number of characters) of this item when printed. Note, however, that C will always print the entire number, even if it is larger than the size specified here. This number refers only to the *minimum* size of the item.

The *precision* parameter denotes the number of digits to the right of the decimal point for floating point numbers, the number of significant digits in exponential numbers, or the minimum number of characters to generate from an integer. The *precision* parameter can also be used to limit the number of characters that will be output in a string conversion.

The *modifier* is used for changing the size of a certain variable that will be input or output. The available *modifiers* include **l** (for long numbers) and **h** (for short numbers). Modifiers are used with integer and double variables; where an **int** variable uses **%d**, a long **int** will need **%ld**.

The *type* parameter is the most important part of the specifier, and the only parameter that is required. The following table lists the *type* characters that are available (including the **l** and **h** mentioned previously). In the Format of Output column, the letter *d* refers to a digit, and the letter *c* to a character.

Type Character	Type of Item to Print	Format of Output	Example of printf( )	Example of Output
d	int (decimal)	[-]dddd	("%d",31)	31
ld	long int	[-]dddd	("%ld",31L)	31
hd	short int	[-]dddd	("%hd",31)	31
i	int	[-]dddd	("%i",31)	31
o	unsigned int	dddd	("%o",31)	37
	(as octal)		("%#o",31)	037
u	unsigned int	dddd	("%u",31)	31
lu	unsigned long int	dddd	("%lu",31L)	31
x	unsigned int	dddd	("%x",31)	1f
	(as hexadecimal)		("%#x",31)	0x1f
X	unsigned int	dddd	("%X",31)	1F
	(as hexadecimal)		("%#X",31)	0X1F
f	double	[-]ddd.ddd	("%5.2f",31.44)	31.44
e	double (exponent)	[-]d.ddde+-dd	("%.2e",123.5)	12.35e+01

Type Character	Type of Item to Print	Format of Output	Example of printf( )	Example of Output
E	double (exponent)	[-]d.dddE+-dd	("%.2E",123.5)	12.35E+01
c	unsigned char	c	("%c",'A')	A
			("%c",65)	A
s	array of char	ccccc	("%s","Joe")	Joe
%	none	%	("%%")	%

Note: The specifiers **%g** and **%G** are also available. Specifier **%g** acts like **%f** or **%e**, based on the size of the number's value; **%G** acts like **%F** or **%E**, also based on the size of the number's value. In addition, the **%s** specifier, used for strings, can take both a *width* and *resolution* parameter. If you specify **%20.5s**, for instance, **printf( )** would print the first 5 characters of the string in a field 20 characters wide.

Although the examples given in the foregoing table are for **printf( )** output, they are also applicable to **scanf( )** for input. For instance, the following line of code is used to read a string:

```
scanf("%s",str);
```

and this line of code is used to read in a floating point number:

```
scanf("%f",&flt);
```

Special Options for scanf( ) The **scanf( )** function has a special option to help clean up input from users. You can place a space character in the *format* string, and it will cause **scanf( )** to skip any *white space* (spaces, tabs, or carriage returns) entered at that point. This is a useful method, when managing single-character input, of ignoring the carriage return from the previous scan. You may have noticed this option used in previous code examples in this book; it looks like this:

```
scanf(" %c", &ch);
```

Notice that the space precedes the **%c** specifier, and causes the **scanf( )** to read and ignore any unnecessary white space before a valid character is pressed.

Also available for use with **scanf( )** is the asterisk (*) used as a *type* parameter. With **scanf( )**, this means an item should be read, but not saved to any variable. Use this parameter when a string is read, to handle the carriage return at the end of the input line, like this:

```
scanf("%s%*c",&name);
```

This example **scanf( )** statement will read the string into the **name** variable, and read and discard the character entered as the carriage return.

Tip: You can accomplish the same end by using the space option mentioned just above, in the next **scanf( )** call. The space option is more typically found in professional C programs.

A Program Using Various User I/O Format Options Certainly there is an abundance of options available for formatted user I/O; but only a few options are commonly used. Still, great flexibility exists, as shown in the next example, Program P9-3.

P9-3

```
/*
   C program to demonstrate options for formatted I/O.
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
    double  r;
    int     i1, i2;
    char    name[30];

    printf("Hello. Please enter a real number: ");
    scanf(" %lf",&r);

    printf("Here are some ways to print it!\n");
    printf(" %lf    %le    %10.2lf    %.2LE \n\n", r, r, r, r);

    printf("Now enter two integers: ");
    scanf(" %d %d",&i1, &i2);
    printf("Here are some ways to show i1:\n");
    printf("       %d    %o   %#o   %x   %X %#X\n",
           i1,i1,i1,i1,i1,i1);
```

```
    printf("By the way, what was your name? ");
    scanf(" %s",name);
    printf("Thanks, %s, for trying this out!\n",name);
}
```

Here is a sample run of Program P9-3 showing several numeric output formats that use **printf()**. Notice how the different output formats print each of the numbers.

```
C:\CDT\CHAP9>p9-3
Hello. Please enter a real number: 543.76
Here are some ways to print it!
 543.760000     5.437600e+002          543.76    5.44E+002

Now enter two integers: 12 55
Here are some ways to show i1:
       12    14   014   c   C   0XC
By the way, what was your name? John
Thanks, John, for trying this out!

C:\CDT\CHAP9>
```

You've learned how the *format* string in a **printf()** statement can contain many specifiers. Program P9-3 illustrates how **printf()** can also contain other text, as well as the following special character sequences:

Character Sequence	Usage in printf()
\n	New line (as discussed throughout the book thus far)
\r	Carriage return (return to left edge of screen)
\t	Moves to next horizontal tab location
\v	Moves down to next vertical tab location (seldom used, except in conjunction with certain printers that allow the setting of vertical tabs)
\000	Inserts a character with octal notation 000
\0	NULL character (ASCII 0)
\b	Inserts a backspace character, thus moving cursor one space to the left

Character Sequence	Usage in printf( )
\\	Inserts a backslash character
\"	Inserts a double-quote character
\'	Inserts a single-quote character

INPUT AND OUTPUT FROM AND TO STRINGS

The previous section discussed functions to read and write information from and to the user of your programs. A similar set of functions is available for reading and writing from and to strings. This technique is often valuable in converting strings into numbers, or numbers into strings.

FORMATTED I/O: SSCANF( ) AND SPRINTF( )

The two functions that work with string I/O are **sprintf( )** and **sscanf( )**. As you might imagine, these routines are patterned after **printf( )** and **scanf( )**, and therefore use the same formatting options. However, as you can see in their prototypes, there is an additional parameter: the string from which data is read, or to which it is written.

```
#include <stdio.h>

int sprintf( char *s, const char *fmt, ... );
int sscanf( const char *s, const char *fmt, ... );
```

The first parameter in a **sprintf( )** call is the string into which the data is to be written. A call like this

```
char outline[80];

sprintf(outline, "This is %d times, %s!\n", 10, "Charlie");
```

will not output anything to the screen. However, the character array **outline** now contains "This is 10 times, Charlie!\n". On the other hand, the following example:

```
char outline[] = "This is 10 times, Charlie!";
char one[10], two[10], four[10], five[10];
```

```
int num;

sscanf(outline,"%s %s %d %s %s",one, two, &num, four, five);
```

assigns these values:

```
one = "This"
two = "is"
num = 10
four = "times,"
five = "Charlie!"
```

Here you see a powerful data conversion routine at work. The command line parameters are all strings, but using **sscanf( )** we have converted some of the parameters into numbers for processing.

USING DISK FILES

In C, working with data in disk files is not much different from working with user I/O. In fact, the routines shown previously for interacting with the user are actually versions of the routines used for file access.

MOVING AMONG DISK FILES

When working with user I/O, in routines such as **printf( )** and **scanf( )**, C uses a default set of files that are always defined and always open. The names and roles of these files are as follows:

Filename	Role
stdin	Standard input device, usually the keyboard
stdout	Standard output device, usually the screen
stdprn	Standard printer device, usually the printer on LPT1
stderr	Standard error device, usually the screen

These associations are defined in the file stdio.h for use by C programs. These definitions are generally fixed, except that MS-DOS allows redirection from the command line, so stdin and stdout files may sometimes refer to disk files or other devices.

To use files in C, your program must first open them, or create them if they don't yet exist. Then you can read from and/or write to the files. Your program must also close the files. C also has routines that let you delete and rename files.

The fopen( ), fclose( ), and freopen( ) Functions

Files in C are often referred to as streams. A *stream* is a sequence of characters that is received from a file or from the user. A stream can also be used for output. When you see streams mentioned in the sections that follow, remember that they are the same as files.

The prototypes for **fopen( )**, **fclose( )**, and **freopen( )** are

```
#include <stdio.h>

FILE *fopen( const char *filename, const char *mode);
int fclose( FILE *stream );
FILE *freopen( const char *filename, const char *mode, FILE
            *stream);
```

As you might guess, **fopen( )** and **fclose( )** perform the work needed to open and close files. The **fopen( )** call returns a pointer to a FILE variable. FILE is a type defined in the stdio.h file for referring to stream files, in which input and output are buffered. The *mode* parameter refers to how the file should be opened, and has one of the values shown here:

Mode Character	File Characteristics
"r"	Read-only access
"w"	Write access starting at the beginning of the file (creates a new file if necessary)
"a"	Append (write access starting at the end of the file)

There are also two modifiers that can follow the *mode* character, in any combination:

- The **b** modifier denotes that the file is to be opened in binary mode. This means carriage returns are not translated into carriage return/line feed pairs, and the end-of-file character is not translated (which can

cause an early end-of-file condition), as would normally be done in text files.

- The + modifier denotes that a read-only file may also be written to, or that a write-only file may also be read.

Here are some examples of combinations of *mode* character modifiers:

***Mode* Character Modifiers**	**File Characteristics**
"rb"	Read-only access; binary file
"r+"	Read and write access
"w+b"	Write and read access; binary file
"ab"	Append to end of file allowed; binary file

Warning: Opening a file in write mode, using "w" or "w+", will create a file if it does not exist. However, if the file does exist, *it will be erased!* Use the "w" mode character modifier with caution, so you don't accidentally erase your files.

If the **fopen( )** is successful, a FILE pointer is returned. Otherwise, a NULL pointer is returned. The stdio.h file provides a value called NULL for determining if these calls were successful; NULL is generally defined as a pointer to zero.

The **fclose( )** function requires the FILE variable as the only parameter. The file is then closed, and further attempts to use it within your program will return undefined results. If a file is not closed before the program ends, and changes have been made to that file, unpredictable results will occur. Be sure to close all files that you open, for safety's sake.

The **freopen( )** function is used to close one file and open another, using the same FILE variable. If the open is successful, the FILE pointer is returned; otherwise, a NULL pointer is returned. The FILE pointer indicates a file that is closed when the function is entered. The new file, if it can be opened, is then given the same FILE pointer. The **freopen( )** function is often used to redirect stdin and stdout.

The MS-DOS implementation of translated (nonbinary) text files follows these guidelines:

- The newline character (\n) ends each line in C.

- Newline and linefeed are the same character (\n).
- MS-DOS ends lines with carriage return/linefeed characters (\r\n). This is read as just a linefeed in translated mode. Thus your C programs need only look for \n at the end of each line, rather than both \n and \r.
- The end-of-file character is Ctrl-Z in MS-DOS (ASCII 26).
- A text file end-of-file character may be different from the physical end-of-file. If the file contains a Ctrl-Z, and the file is opened in translated (nonbinary) mode, no more reads are possible after the Ctrl-Z is read. If the file is opened in binary mode, the file can be read past the Ctrl-Z, all the way to the end of the actual physical file.
- Binary files can contain \n, \r, and Ctrl-Z characters, which can be read in untranslated (binary) mode.

The fseek( ) and ftell( ) Functions

The standard C library provides **fseek( )** and **ftell( )** for moving within a file and determining the current location of the file pointer (an internal pointer that tracks where the next read or write will occur within the file). The prototypes for these calls are as follows:

```
#include <stdio.h>

int fseek( FILE *stream, long int offset, int where );
long int ftell( FILE *stream );
```

The **ftell( )** function returns the current byte offset within the file specified. The returned value is a long integer.

> **Remember:** With nonbinary files, **ftell( )** may not return the actual current position in the file, because carriage return/linefeed pairs (\r\n) are translated into single carriage returns (\n) in translated mode. In binary mode, **ftell( )** will always be correct.

The **fseek( )** function takes three parameters: the FILE pointer variable associated with the file; the distance that you wish to move within the file; and the location from where the movement will be made. The stdio.h file defines the following three valid values for the third parameter (location).

Where Value	Location from Which Movement Begins
SEEK_SET	From the beginning of the file
SEEK_CUR	From the current location
SEEK_END	From the end of the file

When using SEEK_SET with **fseek( )**, a positive offset moves toward the end of the file, starting at the beginning of the file. A positive offset for SEEK_END moves toward the beginning of the file, starting from the end. Both positive and negative values are allowed when using SEEK_CUR, which starts from the current location in the file and moves toward the end of the file with positive values, and toward the beginning of the file with negative values.

The fflush( ) and rewind( ) Functions

With many operating systems (including MS-DOS), file I/O is buffered. This means that large portions of the file may be read when you only ask for a small piece. Later calls for more of the file are then read from memory, instead of waiting for the (comparably) slow disk drive. Also, writing is done to the buffer, and only to the disk when the buffer is full; this speeds up output. The **fflush( )** function is designed to clear the output buffers by writing them to the disk, even if they're not yet full. This is always done before a file is closed by **fclose( )**, but should also be performed at other critical points in your program to be sure no data is lost.

Here is the prototype for **fflush( )**:

```
#include <stdio.h>

int fflush( FILE *stream );
```

The only parameter is the name of the file to flush. Notice, however, that if a NULL pointer is passed in, all the currently open files will be flushed.

Use the **rewind( )** function when you need to open files for sequential access and later need to reset to the beginning of the file. This is the same as using an **fseek( )** to the beginning of the file, as is shown here with the prototype for **rewind( )**:

```
#include <stdio.h>

void rewind( FILE *stream );  /* Prototype */
```

```
/* Equivalent call */
fseek( stream, 0L, SEEK_SET);
```

The remove( ) and rename( ) Functions

The standard C libraries provide two routines for file maintenance. These are **remove( )** and **rename( )**, which have these prototypes:

```
#include <stdio.h>

int remove( const char *filename);
int rename( const char *old, const char *new);
```

The **remove( )** function deletes a file by name. If the function succeeds, a value of zero is returned; otherwise, a nonzero return value results.

The **rename( )** function changes the name of an existing file on the disk. This function takes two parameters: the old and new names for the file. Again, a return value of zero denotes that the function completed successfully.

The tmpfile( ) and tmpname( ) Functions

The two functions **tmpfile( )** and **tmpname( )** are used to create temporary files for use within your programs. Their prototypes are as follows:

```
#include <stdio.h>

FILE *tmpfile( void );
char *tmpname( char *s );
```

The **tmpfile( )** function creates a temporary file on the disk, with a new name. It returns a FILE pointer variable that may then be used in any file I/O functions. When the file is closed or the program ends, the file is removed from the disk (deleted).

In contrast, **tmpname( )** just returns a temporary name. If the **s** parameter is not a NULL pointer, the name is copied into that location. In any case, a pointer to the name created is returned by the function. This function is useful when you need a temporary file and do not want it deleted when the program ends.

FILE INPUT AND OUTPUT

The functions you've explored in the foregoing section let you open, close, and move among disk files. Now you need to learn how to read and write the information in those files. Luckily, these routines are nearly identical to those used with user and string I/O.

The feof( ) Function

One element of file I/O not seen in user I/O is the function **feof( )**, which is used to determine if you have reached the end of a particular file. Here is the prototype:

```
#include <stdio.h>

int feof( FILE *stream );
```

The value returned by **feof()** is either zero, meaning that you have not reached the end of the file specified, or nonzero, denoting that you have indeed reached the end of the file.

> **Note:** The return value of **feof()** should be checked within any loop that reads a file, because you may be able to continue reading after the end of the file, but the data returned will be of no use.

In translated (nonbinary) files, **feof()** returns a nonzero value if the end-of-file character (Ctrl-Z) has been reached. In a binary file, this does not happen until the physical end of the file is reached. Therefore, opening a file in the wrong mode may cause incorrect results from **feof( )**.

The fgetc( ) and fputc( ) Functions

The **fgetc( )** and **fputc( )** functions are used to input and output single bytes (characters) from and to a file. These functions are similar to **getchar( )** and **putchar( )**, the user I/O versions of these routines. The following prototypes show the major difference between the file I/O and user I/O versions: a parameter specifying the file to use.

```
#include <stdio.h>

int fgetc( FILE *stream );
int fputc( int c, FILE *stream );
```

These file I/O routines are also different from their user I/O relatives because **fgetc( )** and **fputc( )** advance the file pointer each time they are called. Thus continuous calls to **fgetc( )** will read all the characters in a file, without the necessity for making calls to **fseek( )**. This is a situation where **rewind( )** is especially useful—once a file has been read or written with **fgetc( )** and **fputc( )**, then **rewind( )** can be used to reset the file to the first byte.

The getc( ) and putc( ) Functions

The **getc( )** and **putc( )** functions are equivalent to **fgetc( )** and **fputc( )**. Their prototypes are as follows:

```
#include <stdio.h>

int getc( FILE *stream );
int putc( int c, FILE *stream );
```

The **putchar( )** and **getchar( )** functions work the same as **putc( )** and **getc( )**, as if using the standard files shown below:

```
#include <stdio.h>

char c;

getchar();    /* The same as getc(stdin);      */
putchar(c);   /* The same as putc(c, stdout);  */
```

The fgets( ) and fputs( ) Functions

These two functions work much as **gets( )** and **puts( )** do, and have the following prototypes:

```
char *fgets( char *s, int n, FILE *stream );
int fputs( const char *s, FILE *stream );
```

These routines are used to read and write character strings from and to files. The **fgets( )** routine reads a string of characters into **s**, until *n*–1 characters have been read, or until a carriage return occurs. This is an ideal technique for reading an entire line of a text file.

The **fputs( )** function writes an entire line to a file. A simple program for copying text files might look like Program P9-4.

Note the following difference between the operation of **puts( )/gets( )** and **fputs( )/fgets( )**: **gets( )** does not read the carriage return pressed by the user, but **puts( )** appends the carriage return upon output. On the other hand, **fgets( )** does read the carriage return, and **fputs( )** does not automatically append one. In either case, you end up with carriage returns at the end of each line (as long as they existed in the input, of course).

P9-4

```
/*
   C program to copy a text file using fgets() and fputs().
   By: L. John Ribar, C DiskTutor
*/
#include <stdio.h>

main()
{
   char inName[15], outName[15];
   FILE *in, *out;
   char words[81];

   printf("Enter old filename: ");
   scanf(" %s",inName);
   printf("Enter new filename: ");
   scanf(" %s",outName);

   in = fopen(inName,"r"); /* Open to read */
   if (in == NULL)  /* Be sure the file exists! */
   {
      printf("error - old file %s not found!\n", inName);
      exit(1);
   }
   out = fopen(outName,"w");
   if (out == NULL)
   {
      printf("error - could not create new file %s\n!",
             outName);
      fclose(in);
      exit(1);
   }

    do
    {
        fgets( words, 80, in);   /* Read a line. */
        fputs( words, out);      /* Write it out. */
    } while ( !feof(in) );
```

```
fclose(in);
fclose(out);
printf("\n\ndone.\n");}
```

The fscanf( ) and fprintf( ) Functions

These two functions are identical to their user I/O counterparts, with the addition of a FILE pointer. The prototypes are as follows:

```
#include <stdio.h>

int fprintf( FILE *stream, const char *format, ... );
int fscanf( FILE *stream, const char *format, ... );
```

You may also notice the similarity to **sprintf( )** and **sscanf( )**, with the FILE pointer replacing the destination string used in those calls. The *format* parameter here follows the same conventions as shown for **printf( )** and **scanf( )**. And there is a direct similarity with **printf( )** and **scanf( )**, in which the FILE pointer is assumed to be **stdin** for **scanf( )** and **stdout** for **printf( )**.

```
#include <stdio.h>

printf( ... );  /* The same as fprintf( stdout, ... );  */
scanf( ... );   /* The same as fscanf( stdin, ... );    */
```

The fread( ) and fwrite( ) Functions

Use these two functions to read and write portions of a file that are not single bytes or strings, such as structures and arrays. The prototypes are:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE
            *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

where **size_t** is a type defined in stdio.h, and generally refers to an integer, but is based on the implementation. Both **fread( )** and **fwrite( )** use a *void pointer*, which means they can utilize any type of pointer. The *nmemb* parameter specifies the number of items to read, and thus allows you to read/write more than one block at a time. Each item or block is of the size

specified in the *size* parameter. The final parameter is, of course, the file to access.

The return value is the number of bytes read or written. If this number does not match the number of bytes you wanted to read or write, an end-of-file condition or an error has occurred.

The ungetc( ) Function

The **ungetc( )** function is a special routine that returns a character to a file stream, meaning that it can be reread at a later time. The prototype is

```
#include <stdio.h>

int ungetc( int c, FILE *stream );
```

This function is useful when you look for a character, find it, and then want to do some other processing before actually coming back to use the character. Only one character is guaranteed to be saved, however.

SUMMARY

- Input and output (I/O) are not part of the C language, but I/O functions are provided in the standard C library.
- I/O is similar for users, strings, and disk files.
- User I/O uses **getch( )**, **putch( )**, **gets( )**, **puts( )**, **scanf( )**, and **printf( )**.
- String I/O (input/output within character strings) uses **sscanf( )** and **sprintf( )**.
- Disk or file I/O uses **getc( )**, **putc( )**, **fgets( )**, **fputs( )**, **fscanf( )**, and **fprintf( )**.
- C keeps several files open and available at all times, including **stdin** (standard input, the keyboard), **stdout** (standard output, the screen), **stdprn** (standard printer, the printer on LPT1), and **stderr** (standard error device, usually the same as **stdout**).

C H A P T E R

DETAILED EXAMPLE: OUTPUT CONTROL

This chapter presents a detailed example of how output can be sent to the printer, screen, or a disk file with no change in the program code. The selection of output destination is made when the program runs, rather than when the program is written.

The printing program in this chapter, Program P10-1, also called the P program, is developed as a complete example. The detailed example in Chapter 8 (the Screen Library) only managed screen displays; the detailed example here, on the other hand, is command-line driven. This means all the user's commands will be given when the program starts.

THE DESIGN SPECIFICATION

Program P10-1 prints out text files to your printer, to the screen, or to a disk file. This program does nearly everything the DOS

PRINT program does, creating more attractive printouts in the process. This program has the ability to

- Print one or more files.
- Accept commands entered on the command line.
- Change the printing format to condensed or bold, and then reset the printer when the program finishes.
- Enable or disable header and footer lines printed at the top and bottom of each page. The header (at the top) shows the filename and the page being printed. The footer displays the date and time the file was printed.
- Print line numbers for each line in the file. This is useful in program code, to help locate compiler errors that specify a line number. Line numbers also help in documenting code, letting you refer to lines by number.
- Change the destination of the output. The default is to send the file to the printer, but there are also options for printing to a file or to the screen. This is useful when the printer you want to use is connected to another computer.

THE DESIGN DECISIONS

Because this program is command-line driven, it requires no special screen-handling functions. However, as an aid to the user, the program will provide help if no parameters are given on the command line. The help will follow this format:

```
Usage:   P <commands> <filenames>

Commands:
     /C    Print compressed print
     /B    Print bold print
     /R    Reset to normal print when done
     /H    Print header (toggle)
```

```
/F    Print footer (toggle)
/L    Print line numbers (toggle)
/O:f  Output to file named f
```

Three of the command options in the foregoing help output format are marked "(toggle)." This means they will toggle between enabled and disabled conditions each time the command occurs on the command line. For example, you might print one file with line numbers, a second file with no line numbers, and a third file with footers, by using a command like this:

```
P /L file1.c /L file2.c /F file3.c
```

Because of the size of this program (rather small), all the code will be kept in one file.

The program uses the following prototypes:

```
void header( int *line, char *title );
void footer( int line );
```

where **header( )** prints out the header at the top of the page, if headers are enabled; **footer( )** prints the footer at the bottom, if footers are enabled.

There are several global variables required to hold the status of the options selected. These are the variables' declarations:

```
FILE *outFile = stdprn; /* Default output to printer */
int doFooter = 0;       /* Should we print footer? (No)  */
int doHeader = 1;       /* Should we print header? (Yes) */
int doReset = 0;        /* Should we reset printer when done?
                           (No) */
int doLines = 0;        /* Should we print line numbers?
                           (No) */
int doCompress = 0;     /* Should we use compressed print?
                           (No) */
int doBold = 0;         /* Should we print with boldface?
                           (No) */
```

Notice that each of the variables above has a default value. The comments state whether the option is enabled or not. The output file (outFile) is assigned to the standard printer (stdprn), but the user can change this with the */O:f* command.

THE SOURCE CODE

The source code for Program P10-1 is fairly straightforward—it's basically a big loop. Each time through the loop, the program reads another command-line parameter. If the first character of the parameter is a slash (/), that parameter is processed as a command. Otherwise, the parameter is treated as a filename, and the program attempts to print the file to the specified destination.

NOTABLE ELEMENTS OF PROGRAM P10-1

Let's look at some special coding techniques used in this program. First, to toggle the header, footer, and line-numbering options, the following statements are used:

```
doHeader = (doHeader==1 ? 0 : 1);   /* toggle */

doFooter = (doFooter==1 ? 0 : 1 ); /* toggle */

doLines = (doLines==1 ? 0 : 1 );    /* toggle */
```

These statements use the ? operator to check the status of the variables **doHeader**, **doFooter**, and **doLines**. If the result is True (==1), then the first value (0, shown before the colon) is assigned; otherwise, the second value (1, listed after the colon) is used. Therefore, if any of these variables has a value of 1 when it enters the statement, the variable is assigned to 0. Otherwise, it is assigned to 1. This is all it takes to create a toggle mechanism.

To change output destinations, we use the following piece of code:

```
strcpy( fileName, &argv[i][3] );
if (outFile != stdprn)
   fclose( outFile );
outFile = fopen( fileName, "w" );
break;
```

In this segment of the program, the filename is first copied to the variable **fileName**. Notice that the filename starts at position 3 in the command, after the **/O:*f*** command. Therefore, the address of that position in the argument string (**&argv[i][3]**) is used as the source of the filename.

The [i] indicates the index for the command-line argument the program is checking.

The next step in the output destination determination is to be sure that another file isn't already open. To do this, the program checks to see that the current output file (outFile) is not equal to the standard printer file (stdprn). If these two are different, then the output file (outFile) is closed, and the new file for output is opened, also using outFile.

To send the printed files to the screen, rather than to the printer or a disk file, run the program with CON: as the output name, as shown below. (CON: is the DOS name for the screen.)

```
p /o:con: file.c
```

Of course, this will probably happen too quickly for you to read the output on the screen. If the CON: feature will be used often, you might consider modifying the program to accept another parameter (/S for screen, perhaps) that will output the file to the screen 22 lines at a time (the average number of rows you will want to display on a screen), and then let the user press a key after each page has been read. Or, read the case study at the end of this chapter for a program more suited to screens.

This next segment of code sends commands to the printer to turn on compressed and bold printing:

```
if (doCompress)
   fprintf( outFile, "%c", 15 );   /* Control-O */
if (doBold)
   fprintf( outFile, "%cE", 27 ); /* ESCape E */
```

These printer commands are specifically for use with most Epson- and IBM-compatible dot-matrix printers. If you have a different type of printer, change the commands to match what your printer requires. For instance, if using the Hewlett-Packard DeskJet 500, you would use these commands:

```
if (doCompress)
   fprintf( outFile, "%c(s16.67H", 27 );  /* Escape sequence */
if (doBold)
   fprintf( outFile, "%c(s1Q", 27 );      /* Escape sequence for
                                             draft mode since
                                             Bold is default */
```

You must also change the **doReset** command to match your printer, as follows:

```
    if (doReset)
       fprintf( outFile, "%c@", 27 );      /* ESCape @ resets
                                              printer */
```

THE PROGRAM

Here is Program P10-1 in its complete form, also known as the P program:

P10-1

```
/*
   Program:     P10.1      (a.k.a. P.C)
   By:          L. John Ribar
                C DiskTutor
   Date:        12/31/92
   Purpose:     This program is designed to print files with
                optional headers and footers. It also allows
                setting of print attributes for the printer.
   Use:         P <commands> <files>
                where
                   <commands> =
                      /C     Print compressed print
                      /B     Print bold print
                      /R     Reset to normal print when done
                      /H     Print header (toggle)
                      /F     Print footer (toggle)
                      /L     Print line numbers (toggle)
                      /O:f   Output to file named 'f'

                   <files> = list of files to print, separated
                             by spaces
*/

/* Preprocessor Commands */

#include <stdio.h>
#include <time.h>
#include <string.h>

#define MAX_LINES 58     /* Maximum lines to print per page */

/* Prototypes */

void header( int *line, char *title );
void footer( int line );
```

```
/* Global Variables */

int Page = 0;
int LineNum;                  /* For printing line numbers */
FILE *outFile = stdprn;       /* Default output to printer */

int doFooter = 0;            /* Should we print footer? (No)  */
int doHeader = 1;            /* Should we print header? (Yes) */
int doReset = 0;             /* Should we reset printer when done?
                                (No) */
int doLines = 0;             /* Should we print line numbers?
                                (No) */
int doCompress = 0;          /* Should we use compressed print?
                                (No) */
int doBold = 0;              /* Should we print with boldface?
                                (No) */

char FooterLine[80];         /* Contents of the footer line */

/* Main Function Code */

void main( int argc, char *argv[] )
{
   FILE *inFile;               /* The file we're reading */
   char inLine[129];           /* The line we just read */
   int Done, start, i, j;      /* Miscellaneous variables */
   int curLine;                /* Current line we're on */
   char fileName[80];          /* Output filename */
   struct tm *time_of_day;     /* Structure for holding time and
                                  date */
   time_t lTime;               /* Time in condensed format */
   char Now[30];               /* Hold the current time string */

   /* Any parameters on the command line? If not, print help.*/
   if (argc < 2)
   {
      printf("Usage:   P <commands> <filenames>\n\n");
      printf("Commands:    \n");
      printf("      /C    Print compressed print\n");
      printf("      /B    Print bold print\n");
      printf("      /R    Reset to normal print when done\n");
      printf("      /H    Print header (toggle)\n");
      printf("      /F    Print footer (toggle)\n");
      printf("      /L    Print line numbers (toggle)\n");
      printf("      /O:f  Output to file named <f>\n");
```

```
      return;        /* Exit program */
   }

   start = 1;

   for (i=start; i<argc; i++)  /* Each command-line parameter */
   {
      if (argv[i][0] == '/')    /* Look for commands */
      {
         switch (argv[i][1])    /* Process the command */
         {
            case 'c':
            case 'C':   /* Change to compressed print */
               doCompress = 1;
               break;
            case 'b':
            case 'B':   /* Change to bold print */
               doBold = 1;
               break;
            case 'r':
            case 'R':   /* Reset mode when finished */
               doReset = 1;
               break;
            case 'h':
            case 'H':   /* Print headers */
               doHeader = (doHeader==1 ? 0 : 1);  /* toggle */
               break;
            case 'f':
            case 'F':   /* Print footers */
               doFooter = (doFooter==1 ? 0 : 1 ); /* toggle */
               break;
            case 'l':
            case 'L':   /* Print line numbers */
               doLines = (doLines==1 ? 0 : 1 );   /* toggle */
               break;
            case 'o':
            case 'O':   /* Output to a file */
               strcpy( fileName, &argv[i][3] );
               if (outFile != stdprn)
                  fclose( outFile );
               outFile = fopen( fileName, "w" );
               break;
            default:
               printf("[%s] is an illegal command!\n",
                     argv[i] );
```

```
            break;
        }
    }
    else      /* Must be a file name */
    {
        inFile = fopen( argv[i], "r" );

        if (inFile == (FILE *)0)
        {
            printf("ERROR! Could not read file %s\n",
                argv[i]);
            continue;  /* Jumps to next i in for loop */
        }
        curLine = 0;
        LineNum = 0;
        Page = 0;

        /* Let the user know what is happening. */
        printf("Printing %s ...", argv[i]);

        /* If needed, set up footer, which includes date and
            time. */
        if (doFooter)
        {
            time( &lTime );
            time_of_day = localtime( &lTime );
            strcpy( Now, asctime( time_of_day ) );
            Now[24] = 0;    /* Removes carriage return */
            strcpy( FooterLine,
                "-- <C DiskTutor> ------------------------ ");
            strcat( FooterLine, Now );  /* Concatenate (add)
                                           to the */
            strcat( FooterLine, " --"); /* end of the footer
                                           line. */
        }
        if (doCompress)
            fprintf( outFile, "%c", 15 );  /* Control-O */
        if (doBold)
            fprintf( outFile, "%cE", 27 ); /* ESCape E */

        fgets( inLine, 128, inFile );
        Done = 0;
        for (;;)
        {
            if (feof(inFile)) Done = 1;
```

```
            /* Get rid of trailing carriage return. */
            if (inLine[strlen(inLine)-1] == '\n' )
               inLine[strlen(inLine)-1] = 0;

            header(&curLine, argv[i]);

            if (doLines)
               fprintf( outFile, "%4d) %s\n", ++LineNum,
                        inLine );
            else
               fprintf( outFile, "%s\n", inLine );
            curLine++;
            footer(curLine);
            if (Done)
               break;
            inLine[0] = 0;
            fgets( inLine, 128, inFile );
         }
         fclose( inFile );    /* Done with this file */
         if (curLine <= MAX_LINES)
         {
            for (j=curLine; j<MAX_LINES; j++)
               fprintf( outFile, "\n");
            footer( MAX_LINES + 2 );
         }
         printf(" done. %d page(s).\n",Page);
      }
   }
   if (doReset)
      fprintf( outFile, "%c@", 27 );    /* ESCape @ resets
                                           printer */
   if (outFile != stdprn)
      fclose( outFile );
}


void header( int *line, char *title )
{
    int i;
   if ((*line > MAX_LINES)||(Page == 0))
   {
      if (doHeader)
      {
         fprintf( outFile, "File: %s ", title );
```

```
            for (i=0; i<60-strlen(title); i++)
               fprintf( outFile, "-");
            fprintf( outFile, " Page: %4d\n\n\n", ++Page );
            *line = 5;
         }
         else
         {
            fprintf( outFile, "\n");
            *line = 3;
         }
      }
   }

   void footer( int line )
   {
      if (line > MAX_LINES)
      {
         if (doFooter)
            fprintf( outFile, "\n%s\n", FooterLine);

         /* Always need a formfeed at the end of the page. */
         fprintf( outFile, "%c", 12 );   // formfeed
      }
   }
```

Case Study 2: A File-dump Utility

Now that you have learned about working with files, try this case study. Develop a program, called an appropriate name such as DUMP, that will read a disk file and dump (send) it to the screen or to a file, in ASCII (text) or hex mode. In addition, your DUMP program should be able to

- Send output to the screen, the printer, or a disk file.
- Specify multiple files on the command line.
- Toggle ASCII and hex mode for each file, using command-line parameters.
- If the file is dumped to the screen, allow the user to stop the display after each screenful of data, in order to read what is being displayed.

Case Study 2: A File-dump Utility (*continued*)

And here are some optional capabilities you may want to include in your DUMP program. These suggestions for optional functions are here just to get you started—use your imagination, and write the program in a way that will be useful to *you* and your work!

- Allow the user to move forward and backward through the file when viewing it on the screen.
- Use the Screen and TUI Library developed in Chapter 8 to create an interactive environment, allowing selection of filenames from within the program; display of the files in a box on the screen (use **Scroll( )** to move up and down); and special keys for commands (arrow keys to move around, function keys to select viewing mode, and so forth).

Remember: If you want to use the Screen Library functions, you must include the "screen.h" header file in your program, and add **screen** to the compiler command line, like this:

```
wcl dump screen
```

This will compile **dump.c** and **screen.c**, and then link them together to create the executable file **dump.exe**.

One way of completing the project in this case study is presented in Chapter 14. As you work on your program, remember to follow the design steps outlined in Chapter 2; determine all that you want the program to do before writing any code. Once your program is completed, be sure to write a one- or two-page document describing how to use the program.

Case Study 2: A File-dump Utility (*continued*)

Figure 10-1 shows what the output of your DUMP program will be if it sends Program P10-1 to the screen in ASCII (text) mode.

Text display (ASCII mode)

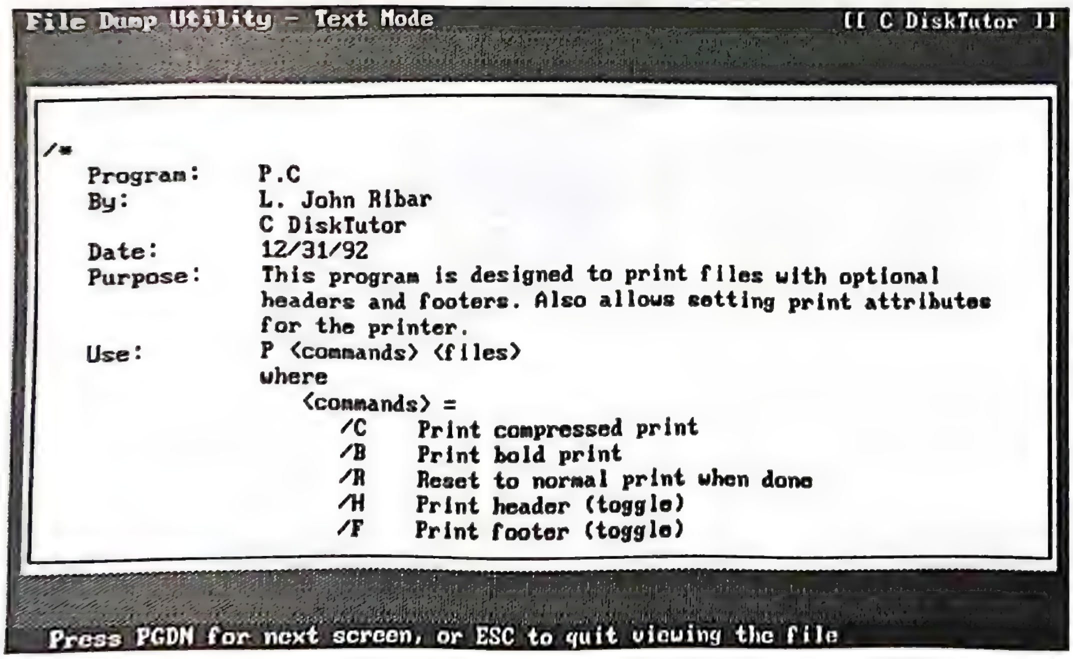


FIGURE 10-1

Text output of Program P10-1 from your DUMP program

Case Study 2: A File-dump Utility (*continued*)

Figure 10-2 shows what the output of your DUMP program will be if it sends Program P10-1 to the screen in hex mode.

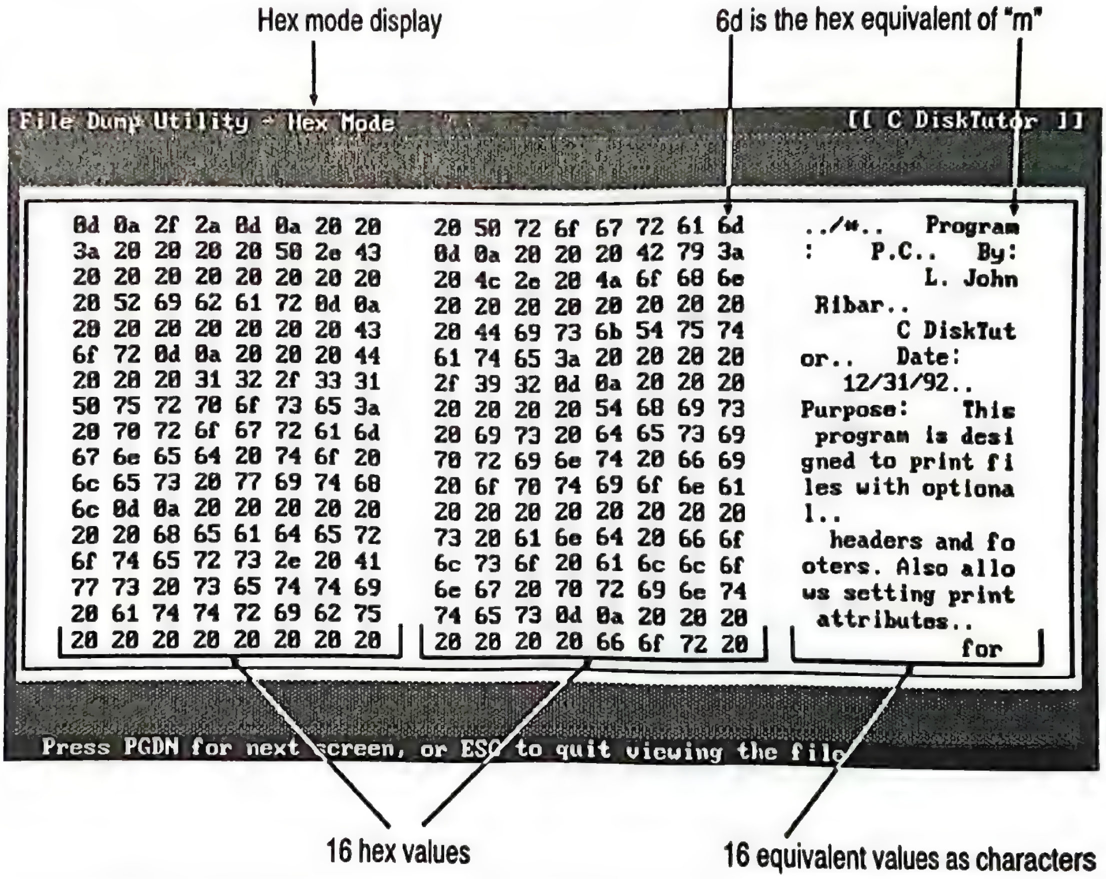


FIGURE 10-2

Hex and character output of Program P10-1 from your DUMP program

C H A P T E R

THE C PREPROCESSOR, HEADER FILES, AND STANDARD C LIBRARY

At this point in the book you are able to create small programs. As you begin to expand your knowledge and learn how to create even larger programs, one important technique to remember is to divide larger programs into functional segments. Functions that you use often can then be grouped into separate files. C's header files, described later in this chapter, make it easier to reuse these common functions.

Some common routines come with standard C compilers, and are part of the standard C library described in the last part of this chapter.

First, however, you'll need a thorough understanding of the C Preprocessor, which is used to create macros and to conditionally compile certain parts of your programs, based on rules you specify. The Preprocessor is also used to include the required header files

that are necessary for your program to run. So let's begin this chapter with a study of this important element of C programming.

PREPROCESSOR DIRECTIVES

C compilers all have a *Preprocessor*. The Preprocessor handles special commands called *preprocessor directives*, all of which start in the first column of a line, with a # character—for example, **#include**. The most common uses for preprocessor directives are for describing macros, and for conditional compilation of code.

MACROS

Macros are defined to the Preprocessor as follows:

```
#define name function_or_value
```

This section contains actual examples, with explanations, to explain this notation.

Example 1 In the following macros,

```
#define True 1
#define False 0
```

the values for True and False are defined as 1 and 0, respectively. *These are not variables!* The Preprocessor looks through your code and replaces all instances of True with the number 1, before the compiler starts the compilation.

These replacement macros are useful for keeping code readable, and you have seen them used throughout this book. They also help when you are making changes to code. To change the value of True, for instance, you would only have to change it once, and all uses of True would have the new value during the next compilation.

Example 2 In the following macros,

```
#define BEGIN   {
#define END     }
#define IF      if (
```

```
#define THEN     )
#define EQUAL    ==
```

common C structures are replaced with English-language words. This type of macro helps programmers accustomed to another language learn and write in C.

With the five macros shown just above included as preprocessor directives, these two segments of code would have identical results:

```
/* First option, not using the macro definitions */

if (a == b)
{
   printf("a and b were equal \n");
}


/* Second option, using the macro definitions */

IF a EQUAL b THEN
BEGIN
   printf("a and b were equal \n");
END
```

Though using the English-language macro definitions actually changes the look of C code, it enables you to use C in a way that is most meaningful to you.

Tip: Putting these common definition macros into a header file (see the next section) makes them available to all your programs.

Example 3 The following macros are used to replace common procedure calls with C equivalents. Once again, macros like these improve the program's readability. Of course, the compiler will get the full version of the macro to compile.

```
#define NewLine()    printf("\n")
#define SayName()    printf("\n\nMy Name is John! \n\n")
```

These macro replacements are simple examples; longer macros can also be written that save many keystrokes and reduce the chance of typing errors in tricky situations.

The parentheses after the foregoing NewLine and SayName examples are not required, but are used to make the code look like functions are being called. Thus the form is more consistent with actual function calls.

Example 4 Parameters can be passed to macros, just as with functions. This is handy when you need a function that can work on multiple data types. For instance, suppose you want the following function to find the minimum of two integers and return it:

```
int min( int a, int b);
```

If you later wanted to check long integers or floating point numbers, you would need to rewrite the entire function. Use a macro, however, and you remove the need for type checking of this sort. Here is an example:

```
#define min( a, b )    ( a<b ? a : b )
```

This macro takes two parameters, and then uses the ? operation. The function can be read as follows: "Is *a* less than *b*? If so, return the value of *a*; otherwise, return the value of *b*." In actual use, the macro will be placed in your C program like this:

```
printf(" The minimum number is %d \n", min(a,b) );
```

but would be translated for the compiler like this:

```
printf(" The minimum number is %d \n", ( a<b ? a : b ) );
```

You can see the savings in typing that results from using macros. Notice, also, how much more readable the code is.

Tip: As you create macros that you use often, collect them in a header file, and use the **#include** directive to read them into your programs.

Undefined Macros Macros can also be undefined. This is useful when the macros are only needed in parts of your programs. To undefine a macro, use the directive **#undef** along with the name of the macro. This is often done with the **#ifdef** directive to test for the previous definitions of macros. (The "Conditional Compilation" section, next, explains **#ifdef**.) The following is an example:

```
#ifdef True
#undef True
#define True 1
#endif
```

This example checks to see if True is already defined. If so, True is then undefined and redefined in the way that is required.

CONDITIONAL COMPILATION

There are times when parts of a program are needed, and times when they aren't. For instance, in a program to handle colors, you need to plan for the maximum number of colors. A monochrome system uses two colors (black and white); a CGA graphics system uses four colors; and an EGA system boasts sixteen. *Conditional directives* can be used to select the maximum number of colors, based on the type of program being written.

Consider this example:

```
#ifdef EGA
#define MaxColors 16
#endif

#ifdef CGA
#define MaxColors 4
#endif

#ifdef Monochrome
#define MaxColors 2
#endif
```

The directives used here are **#ifdef** and **#endif**. The **#ifdef** works like an **if** statement in the C language, testing whether an item is defined. If it is defined, statements are executed until the **#endif** directive occurs.

Unfortunately, the above code segment is somewhat long. To shorten it, you can use a version of the **else** statement, **#else**, as shown in the following example. Notice how each **#ifdef** statement has a matching **#endif**.

```
#ifdef EGA                /*  <-----------\     */
#define MaxColors 16      /*              |     */
#else                     /*  <-----------+     */
#ifdef CGA                /*   <--------\ |     */
#define MaxColors 4       /*            | |     */
```

```
#else                          /*   <--------+   |     */
#ifdef Monochrome              /*    <----\  |   |     */
#define MaxColors 2            /*         |  |   |     */
#endif                         /*    <----/  |   |     */
#endif                         /*   <--------/   |     */
#endif                         /*  <-------------/     */
```

At first glance, this may not seem to be much of a reduction. In other cases, however, you can achieve great savings in the number of lines in the code, or at least a simplification of it.

Another version of the **#ifdef** directive is **#ifndef**, which determines if the item is defined, and if not, performs the necessary statements. Here is an example:

```
#ifndef True
#define True 1
#endif
```

And finally, here is one other way to code the color-determination code example, involving the **#if**, **#elif** (a combination of **#else** and **#if**), and **#endif** directives:

```
#if defined(EGA)
#define MaxColors 16
#elif defined(CGA)
#define MaxColors 4
#elif defined(Monochrome)
#define MaxColors 2
#endif
```

In this case, separate **#endif** directives are not needed, as the entire set of **#if** and **#elif** statements are considered one statement. Notice the great reduction in code size.

HEADER FILES

The **#include** preprocessor directive tells the compiler to read in header files. The **#include** directive can take one of the following two formats:

```
#include <stdio.h>
#include "myStuff.h"
```

The first format is used with standard files, which are stored with the compiler. These are the header files discussed later in this chapter and that you have seen in earlier examples in this book.

The second format, using quotation marks instead of brackets, is used when you want to directly specify the location of the header file. Look again at the foregoing **#include** example that uses quotation marks; it will look in the current directory. This next example, however, will look in the directory specified:

```
#include "c:\\myWork\\myStuff.h"
```

Note the double backslashes; they are present because a single backslash would be translated by the C string-handling routines, causing the **#include** directive to incorrectly interpret "c:myWorkmyStuff.h" as the file to read.

HIGHLIGHTS OF THE STANDARD C LIBRARY

The rest of this chapter describes important components of the standard C library, as implemented by the C DiskTutor compiler. The header files included with the compiler fall into two categories: ANSI-compatible header files (Table 11-1), and those considered to be extensions (additions) to the standard (Table 11-2).

Caution: The extension header files provided by the C DiskTutor compiler are found in many popular C implementations. Be careful when using them, however, because they are not completely portable from one compiler to the next. Look in your compiler manual for more information about the extensions available.

Note: Not all components of the ANSI library, or of the DiskTutor header files, are covered in this chapter; there are simply too many items. If you wish to look at these files in more detail, they are installed in the \CDT\H directory

Name of Header File	General Use
assert.h	Used with **assert** macro
ctype.h	Declarations for functions that manipulate characters
errno.h	Error codes and a declaration of the variable **errno**
float.h	Handling of floating point numbers
limits.h	Limits and ranges of integers and characters
locale.h	Definitions of locales that can be selected with **setlocale( )**
math.h	Mathematical function and structures
setjmp.h	Declarations for use with the **setjmp( )** and **longjmp( )** functions
signal.h	Declarations related to **signal( )** and **raise( )** functions
stdarg.h	Declarations for macros that handle variable argument lists
stddef.h	Declarations for NULL, **size_t**, and other popular constants and macros
stdio.h	Declarations for standard input and output functions
stdlib.h	Declarations of other miscellaneous functions not included elsewhere
string.h	Declarations for string- and memory-block-handling routines
time.h	Declarations for time-handling functions and structures

TABLE 11-1
ANSI-Compatible Header Files

(unless you made changes to the default locations) and listed in Appendix A of this book. A professional compiler package will also include extensive documentation on the use of each function.

Name of Header File	Declarations in the File
bios.h	Those needed for access to PC BIOS routines
conio.h	For console and port I/O
direct.h	For file- and directory-handling
dos.h	Those needed for access to MS-DOS interface
env.h	For environment string-handling functions
fcntl.h	The flags used by **open( )** and **sopen( )** functions, as found in the io.h header
io.h	For functions that perform input and output at the operating-system level
malloc.h	For functions used in allocating and deallocating segments of memory
process.h	Those used in running other processes (programs) from within your programs
search.h	For **lfind( )** and **lsearch( )** functions
share.h	Those required for shared access to files using **sopen( )**

TABLE 11-2
Header Files That Are Extensions to the Standard C Library

TYPE-CHECKING ROUTINES: CTYPE.H

The header file ctype.h contains prototypes for several functions that determine the type of character passed in as a parameter. To use these functions, you need to include the following line in your program, before any of the functions are called:

```
#include <ctype.h>
```

Here are the functions in ctype.h:

The **isalnum( )** function returns a nonzero value (True) if the character *c* is an alphabetic or numeric character (*a* through *z*, *A* through *Z*, or 0 through 9).

```
int isalnum(int c);
```

The **isalpha( )** function returns a nonzero value (True) if the character *c* is an alphabetic character (*a* through *z*, or *A* through *Z*) .

```
int isalpha(int c);
```

The **iscntrl( )** function returns a nonzero value (True) if the character *c* is any control character.

```
int iscntrl(int c);
```

The **isdigit( )** function returns a nonzero (True) value if the character *c* is a valid digit (0 through 9).

```
int isdigit(int c);
```

The **isgraph( )** function returns a nonzero (True) value if *c* is a character that can be displayed when printed on the screen. This includes all ASCII values greater than 32 and less than 128.

```
int isgraph(int c);
```

The **islower( )** function returns a nonzero (True) value if the character *c* is a lowercase letter (*a* through *z*).

```
int islower(int c);
```

The **isprint( )** function returns a nonzero (True) value if the character *c* would receive a True response from **isgraph(c)**, or if *c* is a space character.

```
int isprint(int c);
```

The **ispunct( )** function returns a nonzero (True) value if the character *c* is a punctuation symbol.

```
int ispunct(int c);
```

The **isspace( )** function returns a nonzero (True) value if the character c is one of these space characters: space (' '), form feed ('\f'), new line ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v').

```
int isspace(int c);
```

The **isupper( )** function returns a nonzero (True) value if the character *c* is an uppercase letter (*A* through *Z*).

```
int isupper(int c);
```

The **isxdigit( )** function returns a nonzero (True) value if the character *c* is a digit, that is, if **isdigit(c)** returns True; or if *c* is a valid hex alphabetic character (*a* through *f* and *A* through *F*).

```
int isxdigit(int c);
```

The **tolower( )** function returns the lowercase equivalent of the character *c*, if *c* is an aphabetic character. In all other cases, **tolower( )** returns a copy of *c* without change.

```
int tolower(int ch);
```

The **toupper( )** function returns the uppercase equivalent of the character *c*, if *c* is an alphabetic character. In all other cases, **toupper( )** returns a copy of *c* without change.

```
int toupper(int ch);
```

MATHEMATICAL FUNCTIONS AND DEFINITIONS: MATH.H

The header file math.h contains the prototypes for mathematical functions in the standard C library. To use these routines, you need to include the following line in your program before the actual functions are called:

```
#include <math.h>
```

Note: Notice that the functions in math.h take double (floating point) values, and return double results. If you are not working with double values, you should **cast** the numbers for the most accurate results. For instance, in the following example, you can take the square root of an integer by first casting it to a double:

```
#include <math.h>        /* Include the math header. */

int anInt;               /* Variable declarations */
int result;
```

```
/*
   Cast the parameter anInt to a double, pass it into the
   sqrt( ) function, and then cast the result to an integer
   before assigning the value to the result variable.
*/
result = (int) sqrt( (int) anInt );
```

The functions prototyped in math.h take double precision parameters, and return double precision results.

Errors are reported in the external integer variable **errno**. The error numbers that may be returned are defined in math.h, and include EDOM and ERANGE. EDOM is returned when an input argument is outside the range of values that may be handled by the mathematical function in question. ERANGE is returned when the result of the function cannot be represented as a double value. If the number is too large, ERANGE is stored in **errno**, and HUGE_VAL (defined in math.h) is returned by the function. If the number is too small to be represented, the function returns zero.

Here are the functions in math.h:

The **acos( )** function returns the arc cosine in the range from zero to pi radians. The parameter *x* must be in the range –1 to +1.

```
double acos(double x);
```

The **asin( )** function returns the arc sine in the range from –(pi/2) to +(pi/2) radians. The parameter *x* must be in the range –1 to +1.

```
double asin(double x);
```

The **atan( )** function returns the arc tangent in the range from –(pi/2) to +(pi/2) radians.

```
double atan(double x);
```

The **atan2( )** function returns the arc tangent of *y/x* in the range from –pi to +pi. The signs of the two parameters are used to determine in which quadrant the result will lie. The two parameters cannot both be zero.

```
double atan2(double y, double x);
```

The **cos( )** function returns the cosine of *x*, where *x* is given in radians.

```
double cos(double x);
```

The **cosh( )** function returns the hyperbolic cosine of x.

```
double cosh(double x);
```

The **sin( )** function returns the sine value of *x*, where *x* is given in radians.

```
double sin(double x);
```

The **sinh( )** function returns the hyperbolic sine of *x*.

```
double sinh(double x);
```

The **tan( )** function returns the tangent value of *x*, where *x* is given in radians.

```
double   tan(double x);
```

The **tanh( )** function returns the hyperbolic tangent of *x*.

```
double   tanh(double x);
```

The **exp( )** function returns the exponential value of *x*.

```
double   exp(double x);
```

The **log( )** function returns the natural logarithm of *x*. An error will occur if *x* is negative.

```
double   log(double x);
```

The **log10( )** function returns the base ten logarithm of *x*. An error will occur if *x* is negative.

```
double   log10(double x);
```

The **pow( )** function returns the result of raising *x* to the power *y*. If *y* is not an integral value and *x* is negative, an error will occur. This error will also occur if *x* is zero and *y* is less than or equal to zero.

```
double   pow(double x, double y);
```

The **sqrt( )** function returns the non-negative square root of *x*. A domain error will occur if *x* is negative.

```
double   sqrt(double x);
```

The **ceil( )** function returns the smallest integral value (integer in a double variable) that is larger than *x*. This can be used for rounding a real number up to the closest whole number.

```
double   ceil(double x);
```

The **fabs( )** function returns the absolute value of *x*.

```
double   fabs(double x);
```

The **floor( )** function returns the largest integral number (integer in a double variable) that is smaller than *x*. This can be used for rounding a real number down to the nearest whole number.

```
double   floor(double x);
```

The **fmod( )** function performs a function similar to the % function used with integers; **fmod( )** returns the floating point remainder of *x/y*.

```
double   fmod(double x, double y);
```

GENERAL LIBRARY FUNCTIONS: STDLIB.H

The header file stdlib.h contains the prototypes for several nonspecific functions covering a variety of uses, including integer arithmetic, random numbers, memory allocation, and string-to-numeric conversions. To use these functions, you must place the following line in your program before the functions are called:

```
#include <stdlib.h>
```

Following are the elements of stdlib.h.

As shown below, two **typedefs** are used to define a generic **size** type for use in this file. Type **size_t** is defined as an unsigned integer, and **wchar_t** is defined as an unsigned character.

```
typedef unsigned size_t;
typedef unsigned char wchar_t;
```

These next four functions take as a parameter a pointer to a character string, and process the string to return a double (**atof( )**), integer (**atoi( )**), long integer (**atol( )**), or unsigned long integer (**atoul( )**) value. Each function scans the string until it encounters a character that cannot be used in the number, and then processes the characters read.

```
double   atof(const char *nptr);
int      atoi(const char *nptr);
long     atol(const char *nptr);
unsigned long atoul(const char *nptr);
```

The **abort( )** function is used to exit your program with an error condition. The program stops immediately.

```
void abort(void);
```

The **exit( )** function causes normal program termination. The code that is passed as a parameter is returned to DOS as the program exit code, which can then be checked within batch files.

```
void exit(int code);
```

The **atexit( )** function takes parameters that are the names of functions (without parameters), to be called upon normal program termination, which occurs at the end of the program or after a call to **exit( )**.

```
int atexit(void (*func)(void));
```

The **system( )** function is used to perform a DOS function, as if the function were entered at the DOS command prompt; **system( )** makes use of a secondary copy of COMMAND.COM (the current command processor) to perform its functions.

```
int system(const char *string);
```

For instance, to execute a DIR command, you would use this function call:

```
system("DIR");
```

The **getenv( )** function is used to retrieve the value of the environment variable named by the **name** variable. This is not available under all

operating systems, but does work under MS-DOS and most UNIX implementations.

```
char * getenv(const char *name);
```

The **abs()** function returns the absolute value of integer value *num*. And the **labs()** function returns the absolute value of a long integer.

```
int abs(int num);
long labs(long j);
```

The **div()** and **ldiv()** functions are used to divide integer and long integer values, respectively. The types **div_t** and **ldiv_t** are defined as return values for these functions. Each function performs the operation *numer/denom*, and returns the quotient and remainder in the structure. Program P11-1, following the listing, demonstrates this operation.

```
typedef struct {
  int    quot;
  int    rem;
} div_t;

div_t div(int numer, int denom);

typedef struct {
  long    quot;
  long    rem;
} ldiv_t;

ldiv_t ldiv(long numer, long denom);
```

P11-1

```
/*
   C program to demonstrate integer division.
   By:   L. John Ribar, C DiskTutor
*/
#include <stdio.h>
#include <stdlib.h>

main()
{
    int i1, i2;
    div_t d;

    printf("Enter two integers: ");
```

```
    scanf(" %d %d", &i1, &i2);

    d = div(i1,i2);
    printf("%d divided by %d is %d, with a remainder of %d\n",
        i1, i2, d.quot, d.rem );
}
```

The **malloc( )** function is used to dynamically allocate memory while a program is running. This function takes as a parameter the number of bytes to allocate. The return value is a void pointer, allowing any pointer type to receive the pointer to the allocated memory. The **calloc( )** function dynamically allocates memory in the same manner as **malloc( )**, except as described in the following paragraphs.

```
void    *malloc(size_t size);
void    *calloc(size_t num, size_t size);
```

The difference between **malloc( )** and **calloc( )** is that **calloc( )** works as follows:

- All allocated memory is set to zero.
- Instead of a fixed block size, **calloc( )** has two parameters denoting the size of block to allocate and the number of blocks required.

Thus, to allocate enough memory for 10 integers, you can use either of the following two statements and receive the same results:

```
ptr = malloc ( 10 * sizeof(int) );

ptr = calloc ( 10, sizeof(int) );
```

If a block is allocated using **malloc( )** or **calloc( )**, and later needs to be resized, use a call to **realloc( )**, as shown here:

```
void *realloc(void *block, size_t size);
```

The parameters passed in are the original pointer and the new size. A new pointer is returned, which may or may not point to the original location, based on whether the new size can be accommodated there.

See the function **printPhoneList( )** in Program P15-1 in Chapter 15 for a detailed example of using **calloc( )** for memory allocation.

The **free( )** function is used to release dynamically allocated memory—memory allocated with **malloc( )**, **calloc( )**, and/or **realloc( )**—when it is no longer needed. The only parameter is the pointer to the memory block.

```
void free(void *block);
```

The rand( ) function returns a number from a sequence of pseudorandom numbers generated by the system. The returned number will be between 0 and RAND_MAX (defined in stdlib.h as 32767).

```
int rand(void);
```

The **srand( )** function is used to reseed the **rand( )** function. Using the same seed value each time will result in the same pseudorandom sequence of numbers. Each call to **srand( )** resets the sequence for further calls to **rand( )**.

```
void srand(unsigned seed);
```

STRING-HANDLING FUNCTIONS: STRING.H

The header file string.h contains prototypes for functions that manipulate strings and blocks of memory. The functions that deal with strings (a string of characters ending with the NULL terminator 0) all start with the letters *str*. The functions dealing with blocks of memory (where the size is determined by a parameter, not by a NULL terminator) all begin with the letters *mem*.

To use these functions, include the following line in your program before the actual function calls:

```
#include <string.h>
```

The **strcat( )** function returns the string resulting from the concatenation of the character string pointed to by *dest* and *source*. The pointer returned should be the same as *dest*. The **strncat( )** function works the same way, except that a maximum of *n* characters will be concatenated to *dest*. Notice that if the maximum number of characters are concatenated, there may not be a NULL terminator, which you will have to add.

```
char *strcat(char *dest, const char *source);
char *strncat(char *dest, const char *source, size_t n);
```

The **strcmp( )** function returns the result of comparing the character strings pointed to by *s1* and *s2*. The result is zero if the strings are equivalent, less than 0 if *s1* is less than *s2*, and greater than zero if *s1* is greater than *s2*. The **strncmp( )** function works the same way, except that only a maximum of *n* characters are compared.

```
int    strcmp(const char *s1, const char *s2);
int    strncmp(const char *s1, const char *s2, size_t n);
```

The **strcpy( )** function is used to copy the character string from *source* to *dest*. The final address (*dest*) is returned from the function. The **strncpy( )** function works the same way, except that a maximum of *n* characters are copied. Notice that if the maximum number of characters are copied, there may not be a NULL terminator, which you will have to add.

```
char *strcpy(char *dest, const char *source);
char *strncpy(char *dest, const char *source, size_t n);
```

The **strlen( )** function returns the length of the character string *s*.

```
size_t strlen(const char *s);
```

The **strchr( )** function returns a pointer to the position in *s* where the character *c* is first found. If the character does not occur, a NULL is returned.

```
char *strchr(const char *s, int c);
```

The **strcspn( )** function returns the length of the initial portion of *s1* which does not contain any characters from *s2*. This is an offset into the string, rather than a pointer, as is returned by **strchr( )**.

```
size_t strcspn(const char *s1, const char *s2);
```

The **strstr( )** function returns a pointer to the first location in character string *s1* where character string *s2* is found. If *s2* is not found, a NULL pointer is returned.

```
char *strstr(const char *s1, const char *s2);
```

The functions **memcmp( )**, **memcpy( )**, and **memchr( )** work the same as **strcmp( )**, **strcpy( )**, and **strchr( )**, respectively, except that a block of

memory of *n* bytes is used, rather than a NULL-terminated string. These functions are used in cases where the actual data may contain zero-valued bytes, which should not be considered the end of data.

The **memset( )** function sets *n* bytes, starting at *s*, to the character value *c*. This is used to quickly fill blocks of memory with constant values.

```
int   memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *dest, const void *source, size_t n);
void *memchr(const void *s, int c, size_t n);
void *memset(void *s, int c, size_t n);
```

TIME-HANDLING FUNCTIONS: TIME.H

The header file time.h contains prototypes to manage the use of time. To use these functions in your program, include the following line in your program before the function calls are made:

```
#include <time.h>
```

The following definitions of **time_t** and **clock_t** are used by several of the time-handling routines; these are defined to be unsigned long integers. The **tm** structure is used extensively for passing time information in the time-handling routines.

```
typedef long  time_t;
typedef long  clock_t;

struct  tm   {
  int  tm_sec;        /*  Seconds (0-59)                   */
  int  tm_min;        /*  Minutes (0-59)                   */
  int  tm_hour;       /*  Hours (0-23)                     */
  int  tm_mday;       /*  Day of month (1-31)              */
  int  tm_mon;        /*  Month (1-12)                     */
  int  tm_year;       /*  Years since 1900                 */
  int  tm_wday;       /*  Weekday 0 - 6 (0=Sunday)         */
  int  tm_yday;       /*  Days since Jan 1 (0-365)         */
  int  tm_isdst;      /*  Is it Daylight Savings Time?     */
};
```

Here are the time-handling functions:

The **asctime( )** function returns the string equivalent of the information held in the **tblock** structure.

```
char *   asctime(const struct tm *tblock);
```

Here is an example of the format of the string; the example contains the date and time from the structure **tblock**:

```
Mon Sep 26 01:03:45 1990\n\0
```

Here is the **ctime( )** function; it converts a calendar time, passed in with **timer**, into a string:

```
char *   ctime(const time_t *timer);
```

It is equivalent to using the following statement:

```
asctime( localtime( timer ) );
```

The **clock( )** function returns the approximate amount of processor time that has elapsed since the beginning of your program. This time is in processor ticks; to determine the number of seconds, divide the return value by CLOCKS_PER_SEC (also defined in time.h).

```
clock_t  clock(void);
```

The **difftime( )** function computes the time difference between calendar time *time1* and *time2*.

```
double     difftime(time_t time2, time_t time1);
```

The **localtime( )** function returns a **tm** structure that has been filled correctly from the calendar resident in your computer.

```
struct tm * localtime(const time_t *timer);
```

The **mktime( )** function converts the contents of the structure **timeptr** into the calendar time that is returned. This operation is the inverse of **localtime( )**.

```
time_t   mktime(struct tm *timeptr);
```

The **time( )** function returns the current calendar time. If the pointer **timer** is not NULL, the current calendar time is also assigned there.

```
time_t   time(time_t *timer);
```

AN EXAMPLE

One of the extensions provided by the C DiskTutor compiler (and many others) is the use of graphics functions, made available by the graph.h header file. A detailed discussion of these routines is beyond the scope of this book; however, we recommend you take some time to look through these header files. In fact, here's some incentive: for an exciting demonstration of the graphic features available in the DiskTutor compiler, load and run Program P11-2.

P11-2

```
/*
   Program:    Lines.C

   Purpose:    Demonstrate some of the C Graphics Library
               functions.

   By:         L. John Ribar
               C DiskTutor
*/

/*  Preprocessor Directives  */

#include <graph.h>
#include "screen.h"

#define UNUSED 0
#define LINE   1
#define BOX    2

#define MAX_OBJECTS 50

/*  Local Structures  */

typedef struct {
   int x1, y1, x2, y2;
   int color;
   int shape;
} OBJECT;

/*  Global and Static Variables  */
int nightMode = 0;
int bwMode = 0;
```

```
/*  Prototypes  */

void DrawObject( OBJECT o );
void EraseObject( OBJECT o );
void Change( short *p, short *po, short max );

/*  Main Function Code */

void main( int argc, char *argv[] )
{
   char ch;
   int i, iPos;
   int clr;
   short x1, y1, x2, y2;
   short x1off = 3, x2off = 5, y1off = -2, y2off = 4;
   OBJECT items[MAX_OBJECTS];
   struct videoconfig v;
   short maxX, maxY;

   if (argc >1)
   {
      for (i=1; i<argc; i++ )
      {
         if (argv[i][0] == '/')
            switch (argv[i][1])
            {
               case 'n':
               case 'N':
                  nightMode = 1;
                  break;
               case 'b':
               case 'B':
                  bwMode = 1;
                  break;
               default:
                  break;
            }
      }
   }

   x1off=2;     y1off=4;
   x2off=3;     y2off=5;
   x1 = y1 = 10;
   x2 = 300;    y2 = 250;
   for (i=0; i<MAX_OBJECTS; i++)
```

```
      items[i].shape = UNUSED;

   /* Set the video mode. */
   _setvideomode( _VRES16COLOR );

   /* Read the video configuration. */
   _getvideoconfig( &v );

   /* Determine max X and Y coordinates. */
   maxX = v.numxpixels;
   maxY = v.numypixels;

   iPos = 0;       /* Where in the array? */
   if (bwMode)
      clr = 7;     /* Base color is white. */
   else
      clr = 8;     /* Base color is light gray. */

   for (;;)     /* Forever */
   {
      items[iPos].x1 = x1;
      items[iPos].x2 = x2;
      items[iPos].y1 = y1;
      items[iPos].y2 = y2;
      items[iPos].color = clr;
      items[iPos].shape = LINE;     /* Draw a line. */
      DrawObject( items[iPos] );

      iPos++;
      if (iPos >= MAX_OBJECTS)
      {
         if (bwMode)
         {
            clr = (clr==7)?15:7;
         }
         else if (nightMode)
         {
            clr = 8;    /* Always */
         }
         else
         {
            clr++;
            if (clr >= 16) clr=8;
         }
         iPos = 0;
```

```
        }
        if (items[iPos].shape != UNUSED)
           EraseObject( items[iPos] );

        if (Keypressed())
        {
           ch = Inkey();
           if ( (ch == 'q')||(ch == 'Q') )
              break;
           else
              ch = Inkey();
        }

        Change( &x1, &x1off, maxX );
        Change( &x2, &x2off, maxX );
        Change( &y1, &y1off, maxY );
        Change( &y2, &y2off, maxY );
    }

    _setvideomode( _TEXTC80 );

}

/*  Functions Prototyped Above  */

void DrawObject( OBJECT o )
{
    _setcolor( o.color );
    _moveto( o.x1, o.y1 );
    _lineto( o.x2, o.y2 );
    if (nightMode)
       _setcolor(7);
    else
       _setcolor( 15 );
    _setpixel( o.x1, o.y1 );
    _setpixel( o.x2, o.y2 );
}

void EraseObject( OBJECT o )
{
    _setcolor( 0 );
    _moveto( o.x1, o.y1 );
    _lineto( o.x2, o.y2 );
}
```

```
void Change( short *p, short *po, short max )
{
    *p += *po;
    if (*po < 0)
    {
        if (*p <  0)
        {
            *po *= -1;
            *p += *po;
        }
    }
    else
    {
        if (*p > max)
        {
            *po *= -1;
            *p += *po;
        }
    }
}
```

Case Study 3: An Electronic Address Book

Now that you have learned about working with C header files and the standard C library functions, try developing the program presented in this case study. This program is an electronic name-and-address book; call it an appropriate name, such as BOOK. This project will make extensive use of the Screen TUI Library and the file-handling functions you learned in this and the last few chapters.

Here are the capabilities required:

- Use the TUI Library to include a friendly user interface, as shown in Figure 11-1.
- Allow adding people's names to the data file.
- Allow searching for a name in the data file. Display the record if the name is found, or provide a message if the name is not found.
- Allow printing of mailing labels, or a sorted telephone number listing.
- When printing labels, allow printing of a test label first, so the user can adjust the printer if necessary.

Case Study 3: An Electronic Address Book (*continued*)

- Allow optional printing to the printer, a disk file, or the screen.
- Provide for the deletion of names from the file. Once a record is deleted, it should not show up in printouts or in search results.

Be sure to spend some quality time designing this program before writing the code. There are many interesting ways to perform the functions required. Also, the data file can be in any format you desire; one suggested layout is shown in Figure 11-2.

A possible solution for this case study is presented in Chapter 15.

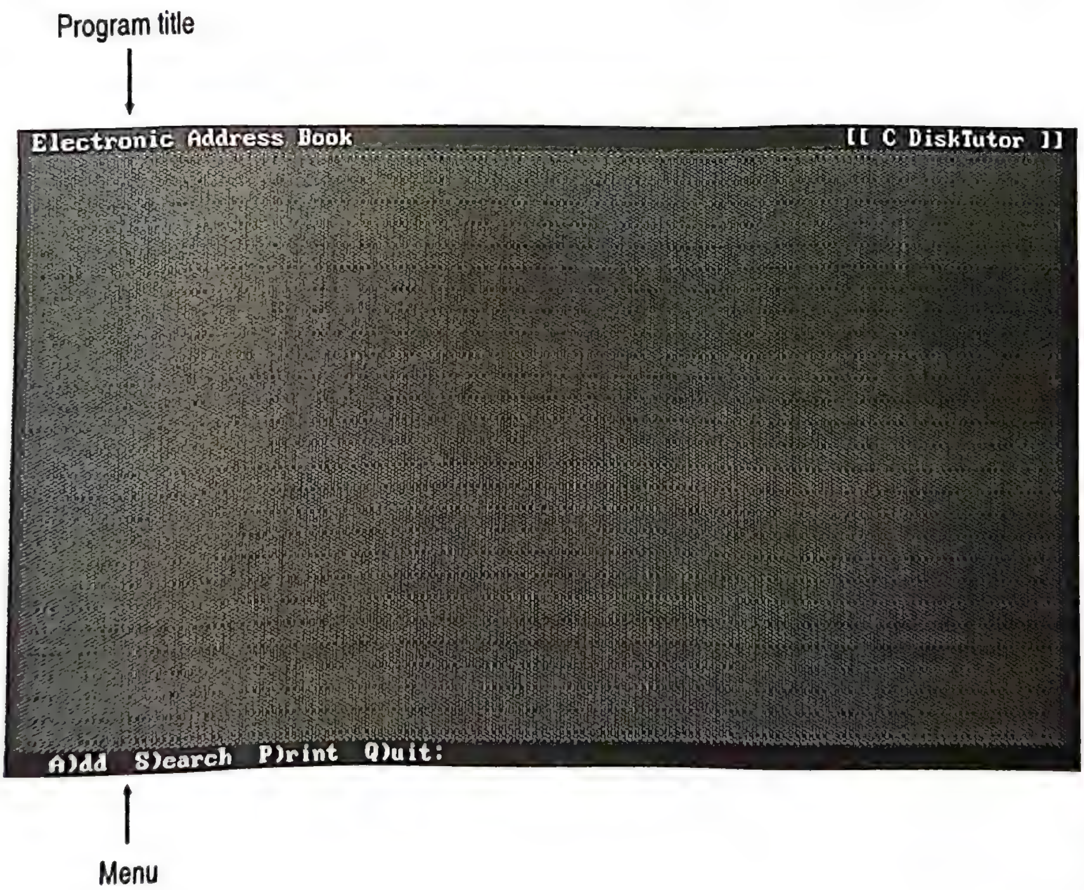


FIGURE 11-1
The opening screen of the electronic address book

Case Study 3: An Electronic Address Book (*continued*)

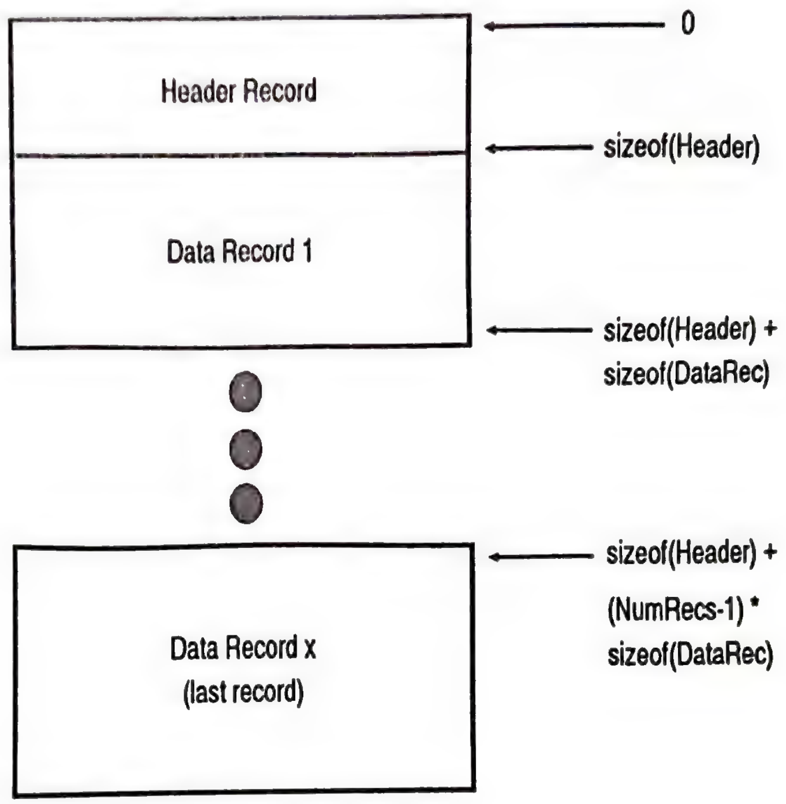


FIGURE 11-2
File layout for the electronic address book

PART THREE

COMPLETING THE CASE STUDIES

C H A P T E R

MINI-CASE SOLUTIONS

The mini-case studies presented throughout this book encouraged you to apply your new knowledge of C to specific small problems. In this chapter, possible solutions for each of the mini-cases are offered. The programs that comprise these solutions can also be found on the disk bundled with this book. To take advantage of the programs on the *C DiskTutor* disk, please note the filenames assigned to each solution and mentioned in the sections to follow. Hopefully, however, you have taken the opportunity to develop your own solutions, so the ones presented here will simply give you additional ideas for enhancements of your own.

MINI-CASE 1

The purpose of Mini-Case 1 in Chapter 4 was to develop a program that reads a number entered by the user at the keyboard, and then prints the value in decimal, octal, and hexadecimal notations. The solution presented here, Program P12-1, is in the file P12-1.C on the *C DiskTutor* disk.

The formatting capability of printf( ) is the major C feature used in this program. The %d, %o, and %x specifiers perform the conversion of a number into decimal, octal, and hexadecimal notation.

P12-1

```
/*
   Mini-Case 1.

   Program:        CVT.C          (a.k.a. P12-1.C)

   Purpose:        Read a number and print it out in several
                   ways.

   By:             L. John Ribar, C DiskTutor
                   January 3, 1993
*/

#include <stdio.h>

main()
{
   int number;     /* The number to display */

   printf("Enter a decimal integer: ");
   scanf( " %d", &number );

   printf("Decimal %d, Octal %o, Hex %x\n", number, number,
          number );
}
```

If you wish to extend this mini-case study a bit, you can also use the %d, %o, and %x specifiers in the call to scanf( ). For instance, Program P12-2 is a second rendition of Mini-Case 1. In this program, you first prompt for the type of number to be entered by the user (decimal, octal, or hex), and then print the value in all three notations.

P12-2

```
/*
   Mini-Case 1, 2nd attempt.

   Program:        CVT2.C         (a.k.a. P12-2.C)

   Purpose:        Read a number in one of three bases,
                   and print it out in all three bases.

   By:             L. John Ribar, C DiskTutor
```

```
                    January 3, 1993
*/

#include <stdio.h>

main()
{
   int number;
   char choice;

   printf("Do you want to enter the number as d)ecimal,
          o)ctal, or h)ex? ");
   scanf( " %c", &choice );

   printf("Enter an integer: ");  /* same prompt for all
                                     three */
   switch (choice)
   {
      case 'd':
      case 'D':
         scanf( " %d", &number ); break;
      case 'o':
      case 'O':
         scanf( " %o", &number ); break;
      case 'h':
      case 'H':
         scanf( " %x", &number ); break;
      default:
         /*
            If the user entered a type that we don't understand,
            force him to enter a decimal number instead.
         */
         printf(" (decimal, please): ");
         scanf( " %d", &number ); break;
   }
   /* While the '0' in front of the octal output, and the "0x"
      in front of the hex output don't actually change the
      values that are printed, these characters make the
      numbers appear more readable as being in the bases
      specified. */
   printf("Decimal %d, Octal 0%o, Hex 0x%X\n", number,
          number, number );
}
```

MINI-CASE 2

The purpose of Mini-Case 2 in Chapter 4 was to develop a program to read the number of minutes entered by the user, and then convert that number into hours and minutes. Once converted, the program needed to print the results.

The solution presented here, Program P12-3, is in the file P12-3.C on the *C DiskTutor* disk. This next program features a major C capability: mathematical calculations on integer numbers, including division and modulus.

P12-3

```
/*
   Mini-Case 2.

   Program:        HOURS.C                  (a.k.a. P12-3.C)

   Purpose:        Convert minutes into hours and minutes.

   By:             L. John Ribar, C DiskTutor
                   January 3, 1993
*/

#include <stdio.h>

main()
{
   int totalMins;

   printf(“Enter total minutes: ”);
   scanf(“ %d”, &totalMins);

   printf(“Total time is %d hours and %d minutes\n”,
         totalMins / 60, totalMins % 60 );
}
```

MINI-CASE 3

There is a function in the standard C library that allows you to get from the system the amount of time your program has been running (in timer

ticks). Mini-Case 3 in Chapter 5 required you to use this function, called clock( ), in a program that waits for a keystroke and then prints the time of delay in seconds.

The solution presented here, Program P12-4, is in the file P12-4.C on the *C DiskTutor* disk. The major C features used in this next program include calling library functions, using functions and #define's from header files in your programs, and performing mathematical functions.

P12-4

```
/*
   Mini-Case 3.

   Program:        TIMER.C         (a.k.a. P12-4.C)

   Purpose:        Print the delay time in seconds, using
                   the clock() function and getchar().

   By:             L. John Ribar, C DiskTutor
                   January 2, 1993
*/

#include <stdio.h>
#include <time.h>

main()
{
   clock_t startClock, nowClock;
   unsigned seconds;
   char ch;

   startClock = clock();
   printf("Press a key to stop the timer ..." );
   ch = getchar();
   nowClock = clock();

   /*
      CLOCKS_PER_SEC is a definition from the time.h header
      file. It tells the number of timer ticks that occurs
      for each second of the real clock. If you divide this
      number into the total ticks since midnight (myClock),
      you will get the number of seconds since midnight.
   */
   seconds = ( nowClock - startClock) / CLOCKS_PER_SEC;
                 /* Total seconds timer was running */
```

```
    /*
       Now display the results.
    */
    printf("\n\nTimer ran for %u seconds\n", seconds );

}
```

MINI-CASE 4

As another version of Mini-Case 3, your fourth mini-case study, offered in Chapter 5, required you to print the delay time in English-language words, for example,

Four minutes and five seconds

instead of

245 seconds

The solution presented here in Program P12-5 is in the file P12-5.C on the *C DiskTutor* disk. Compare it to the program you wrote. The major C features in this program include using arrays of strings, as well as doing some detailed planning and designing of the calculations needed to determine the words to be printed.

P12-5

```
/*
   Mini-Case 4.

   Program:       TIMER2.C          (a.k.a. P12-5.C)

   Purpose:       Print the delay time, using the clock()
                  function, and in English terms.

   By:            L. John Ribar, C DiskTutor
                  January 2, 1993
*/

#include <stdio.h>
#include <time.h>
```

```
/* Global Variables */

char *ones[] = { "zero", "one", "two", "three", "four", "five",
                 "six", "seven", "eight", "nine" };
char *tens[] = { "oh", "ten", "twenty", "thirty", "forty",
                 "fifty", "sixty", "seventy", "eighty",
                 "ninety" };
char *teens[]= { "ten", "eleven", "twelve", "thirteen",
                 "fourteen", "fifteen", "sixteen", "seventeen",
                 "eighteen", "nineteen" };

main()
{
   clock_t myClock, startAt, endAt;
   unsigned hours, minutes, seconds;
   char ch;
   int sayAnd;

   startAt = clock();      /* get starting time */
   printf("Press a key to stop the timer ..." );
   ch = getchar();
   endAt = clock();        /* get stop time */
   myClock = endAt - startAt;

   sayAnd = 0;   /* Don't need to say "and" yet. */
   seconds = myClock / CLOCKS_PER_SEC;    /* total seconds */
   minutes = seconds / 60;                /* total minutes */
   hours = minutes / 60;                  /* total hours */

   /*
      Time is given as hours:minutes:seconds. The hours have
      already been calculated. Now calculate the minutes by
      subtracting the hours from the total time.
   */
   minutes -= (hours * 60);  /* Get rid of the hours. */

   /*
      Now calculate the seconds, by removing the hours and
      minutes from the total time.
   */
   seconds -= ( (hours * 3600) + (minutes * 60) );

   if ( (hours>0) || (minutes>0) )
      sayAnd = 1;
```

```
    /*
       Now display the results in English.
    */
    printf("Total delay was ");

    if (hours >= 10)
       printf("%s hours ", teens[hours-10] );
    else if (hours > 1)
       printf("%s hours ", ones[hours]);
    else if (hours == 1)
       printf("%s hour ", ones[hours]);

    if (minutes > 19)
    {
       if (minutes % 10)
          printf(" %s-%s minutes ", tens[minutes / 10],
                         ones[minutes % 10] );
       else
          printf(" %s minutes", tens[minutes / 10] );
    }
    else if (minutes > 9)
       printf(" %s minutes ", teens[minutes-10] );
    else
       printf(" %s minutes ", ones[minutes] );

    if (sayAnd)
       printf("and ");

    if (seconds > 19)
    {
       if (seconds % 10)
          printf(" %s-%s seconds ", tens[seconds / 10],
             ones[seconds % 10] );
       else
          printf(" %s seconds", tens[seconds / 10] );
    }
    else if (seconds > 9)
       printf(" %s seconds ", teens[seconds-10] );
    else
       printf(" %s seconds ", ones[seconds] );

    printf("\n");    /* Finished! */
}
```

MINI-CASE 5

The purpose of Mini-Case 5 in Chapter 6 was to develop a program called ENCODE.C, which reads a string from the user, asks for a password number, and prints out an encoded copy of the string.

The solution presented here in Program P12-6 is in the file P12-6.C on the *C DiskTutor* disk. The major C feature used in this program is the manipulation of characters within character strings.

P12-6

```
/*
   Mini-Case 5.

   Program:     ENCODE.C        (a.k.a. P12-6.C)

   Purpose:     This program reads a string and a password
                number from the user, and encodes the string
                by adding the number to each character in the
                string.

   By:          L. John Ribar, C DiskTutor
                January 9, 1993
  */

#include <stdio.h>

main()
{
   char inLine[80];
   int i;
   int pswd;

   printf("Enter the password number: ");
   scanf( " %d", &pswd );
   printf("Enter the string to encode on the next line:\n");
   scanf( " %s", &inLine );

   for (i=0; i<strlen(inLine); i++)
   {
      inLine[i] += pswd;
   }
   printf("The encoded string is:\n%s\n\n", inLine );
}
```

MINI-CASE 6

In Mini-Case 6 in Chapter 6, you were to develop a decoder program to go with Mini-Case 5 on the *C DiskTutor* disk. One alternative was to modify Mini-Case 5 to ask whether the user wants to encode (through adding the password) or decode (through subtracting the password) the given string.

The solution presented in Program P12-7 is in the file P12-7.C on the *C DiskTutor* disk. You may notice that it is nearly identical to the solution for Mini-Case 5. In fact, you can use these programs in any order to encode/decode a message—that is, you can decode a text message and have your partner encode it, and achieve results that are the same as if you had encoded the message and your partner had decoded it. Just be sure that the encoder and the decoder know in which order to run the programs.

P12-7

```
/*
   Mini-Case 6.

   Program:    DECODE.C          (a.k.a. P12-7.C)

   Purpose:    This program reads a string and a password
               number from the user, and decodes the string
               by subtracting the number from each character
               in the string.

   By:         L. John Ribar, C DiskTutor
               January 9, 1993
  */

#include <stdio.h>

main()
{
   char inLine[80];
   int i;
   int pswd;

   printf("Enter the password number: ");
   scanf( " %d", &pswd );
   printf("Enter the string to decode on the next line:\n");
   scanf( " %s", &inLine );
```

```
    for (i=0; i<strlen(inLine); i++)
    {
        inLine[i] -= pswd;
    }
    printf("The decoded string is:\n%s\n\n", inLine );
}
```

MINI-CASE 7

In Mini-Case 7, introduced in Chapter 7, you were asked to develop a program called WHENNEXT that reads the name of a day (such as Monday, Tuesday, Wednesday, and so on) from the command line, and displays the date for the next occurrence of that day. Due to the complexity of this mini-case study, the details of its execution are repeated here. If we assume for the purposes of this example that the run command for Friday the 10th of November was

```
WHENNEXT Monday
```

then the expected result would be

```
Monday, November 13
```

For this project you need to use two functions from the standard C library, whose prototypes are as follows:

```
#include <time.h>

time_t time( time_t *tloc );
struct tm *localtime( const time_t *timer );
```

These functions, and the variable types they use, are both declared in the <time.h> header file. Be sure to include this header file in your program.

The **time()** function is used to get the current time, in encoded format, into a variable declared with type **time_t**. The **localtime()** function then converts this encoded time into a **struct**, which is declared as follows:

```
struct tm {
    int  tm_sec;     /* Seconds after the minute -- [0,61] */
    int  tm_min;     /* Minutes after the hour   -- [0,59] */
```

```
   int  tm_hour;   /* Hours after midnight       -- [0,23] */
   int  tm_mday;   /* Day of the month           -- [1,31] */
   int  tm_mon;    /* Months since January       -- [0,11] */
   int  tm_year;   /* Years since 1900                     */
   int  tm_wday;   /* Days since Sunday          -- [0,6]  */
   int  tm_yday;   /* Days since January 1       -- [0,365]*/
   int  tm_isdst;  /* Daylight Savings Time flag */
};
```

Since **localtime( )** returns a pointer to a struct **tm**, you will need a pointer variable to access the information.

One possible solution for Mini-Case 7 is Program P12-8, found in the file P12-8.C on the *C DiskTutor* disk. The C features shown here include system routines and header files, the handling of structures, and the use of pointers to structures.

P12-8

```
/*
   Mini-Case 7.

   Program:    WHENNEXT.C          (a.k.a. P12-8.C)

   Purpose:    Determine the date of the next occurrence of a
               particular day. For instance, if today were
               Friday the 10th, WHENNEXT Sunday would give
               the date Sunday the 12th.

   By:         L. John Ribar, C DiskTutor
               January 2, 1993
*/

/*  Preprocessor Directives  */
#include <stdio.h>
#include <time.h>

/*  Global Variables  */

int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
             };

/*  Main Function Code  */

void main( int argc, char *argv[] )
{
   time_t cTime;
```

```
struct tm *cTm;
int dayWanted = 0;
int thatDay, thatMonth;
int dayOfWeek;

/* If no command line arguments, show some help. */
if (argc < 2)
{
   printf("Usage:  WhenNext <day_of_week>\n");
   return;
}

cTime = time(NULL);
cTm = localtime( &cTime );
thatDay = cTm->tm_mday;
thatMonth = cTm->tm_mon;
dayOfWeek = cTm->tm_wday;

/* Now determine the day requested on command line. */
switch (argv[1][0])
{
   case 's':   /* Saturday or Sunday */
   case 'S':
      if (argv[1][1] == 'u')
         dayWanted = 0; /* Sunday */
      else
         dayWanted = 6; /* Saturday */
      break;
   case 'm':   /* Monday */
   case 'M':
      dayWanted = 1;
      break;
   case 't':   /* Tuesday or Thursday */
   case 'T':
      if (argv[1][1] == 'u')
         dayWanted = 2; /* Tuesday */
      else
         dayWanted = 4; /* Thursday */
      break;
   case 'w':   /* Wednesday */
   case 'W':
      dayWanted = 3;
      break;
   case 'f':   /* Friday */
   case 'F':
```

```
            dayWanted = 5;
            break;
        default:
            printf("I do not know that day!\n");
            return;
    }

    if (dayOfWeek > dayWanted)          /* Day wanted is Next
                                           week */
    {
        thatDay += (7-dayOfWeek) + dayWanted;
    }
    else if (dayOfWeek == dayWanted)   /* One week from today */
    {
        thatDay += 7;
    }
    else                               /* Later this week */
    {
        thatDay += (dayWanted - dayOfWeek);
    }
    if (thatDay > days[thatMonth])
    {
        thatDay -= days[thatMonth];
        thatMonth ++;
        if (thatMonth >= 12)
            thatMonth = 0;
    }
    printf("The date for next %s is %02d/%02d\n", argv[1],
             thatMonth+1,
        thatDay );
}
```

MINI-CASE 8

The purpose of Mini-Case 8 was to change the NOW program (Program P7-9). The NOW program, as you remember, allows you to display the current time. Mini-Case 8 enabled you to change the NOW program so that it prints out whether the current time is in the morning or evening. (The NOW program follows Mini-Case 7 in Chapter 7.) For example, instead of

```
It is now 14:22
```

the program will print

```
It is now 2:22 p.m.
```

The solution presented here in Program P12-9 is in the file P12-9.C on the *C DiskTutor* disk. The C features demonstrated here include **if** statements, the use of pointers to structures, and basic mathematical functions.

P12-9

```
/*
   Mini-Case 8.

   Program:        NOW2.C        (a.k.a. P12-9.C)

   Purpose:        Print the current time, adjusted for
                   morning or evening hours.

   By:             L. John Ribar, C DiskTutor
                   January 2, 1993
*/

#include <stdio.h>
#include <time.h>

main()
{
   time_t nowEncoded;
   struct tm *Now;

   nowEncoded = time( NULL );
   Now = localtime( &nowEncoded );
   if (Now->tm_hour > 12)
      printf("It is now %02d:%02d pm\n", Now->tm_hour-12,
             Now->tm_min );
   else
      printf("It is now %02d:%02d am\n", Now->tm_hour,
             Now->tm_min );

}
```

ENHANCEMENTS

The eight mini-case studies provided in this book are small projects that you can complete to improve your understanding of certain C concepts. However, just as Mini-Case 2 was a suggestion for a second version of Mini-Case 1, you can develop enhancements to the other mini-case study programs. In fact, you are encouraged to do this; in the process you will

- Start thinking in C on your own, as you develop your ideas and design the enhancements.
- Create customized tools that you can use to perform exactly the functions you need accomplished in your daily work on the computer.

C H A P T E R

CASE STUDY 1: A PROGRAMMER'S CALCULATOR

The requirements for Case Study 1, the Programmer's Calculator, were presented in Chapter 8 and are listed again here. In this chapter, you will see the program developed step by step, with additional functions added in small increments. The Programmer's Calculator will be able to

- Run either from the command line, or interactively. The user can enter commands on the command line, in which case the program prints an answer. Alternatively, if the user enters no parameters on the command line, the program will interactively ask the user for numbers and commands, and print the results as they are calculated.
- Receive commands and numbers entered by the user without any mode changing or special handling required. This requires an input function that can read multiple items from the user at one time.

- Print the answer in decimal, octal, and hexadecimal notation, as demonstrated in the command line result shown below.

```
C:\CDT\CHAP13>p13-2 100 * 3 + 14
result is 314 decimal, 0472 octal, 0x13A hex,

C:\CDT\CHAP 13>
```

The following optional assignments for the Programmer's Calculator case study were also presented in Chapter 8. These optional requirements are intended to help you start thinking about your own ideas for enhancements.

- Use the Screen Library developed in Chapter 8 to make the calculator program look more professional. Try making the screen look like the tape on a manual calculator, scrolling the user's entries as they are processed (see Figure 13-1).
- Add memory functions (such as place a number in memory, recall from memory, add to memory, and so on) to the Calculator. Multiple calculator memories would be better than a single memory.
- Allow the user's numbers and commands to be displayed on the screen, and at the same time printed at the printer, for a hardcopy reference.

Certainly the purpose of this case study is to practice programming in C, but it's also intended to help you realize that you can now write your own useful programs in the C language.

STARTING THE PROJECT

Before getting into the screen-management functions, you can start coding this program to handle the command-line parameters. The pseudocode for how this program is designed might look like the following:

```
Initialization
IF There Are Command-line Parameters THEN
   Process Command-line Parameters
   Print the Result
ELSE
   Display the TUI Screen
   REPEAT
      Accept Commands from the User
      Process the Commands
      Display Answers on the Screen
   UNTIL Command = 'Q'
END IF
```

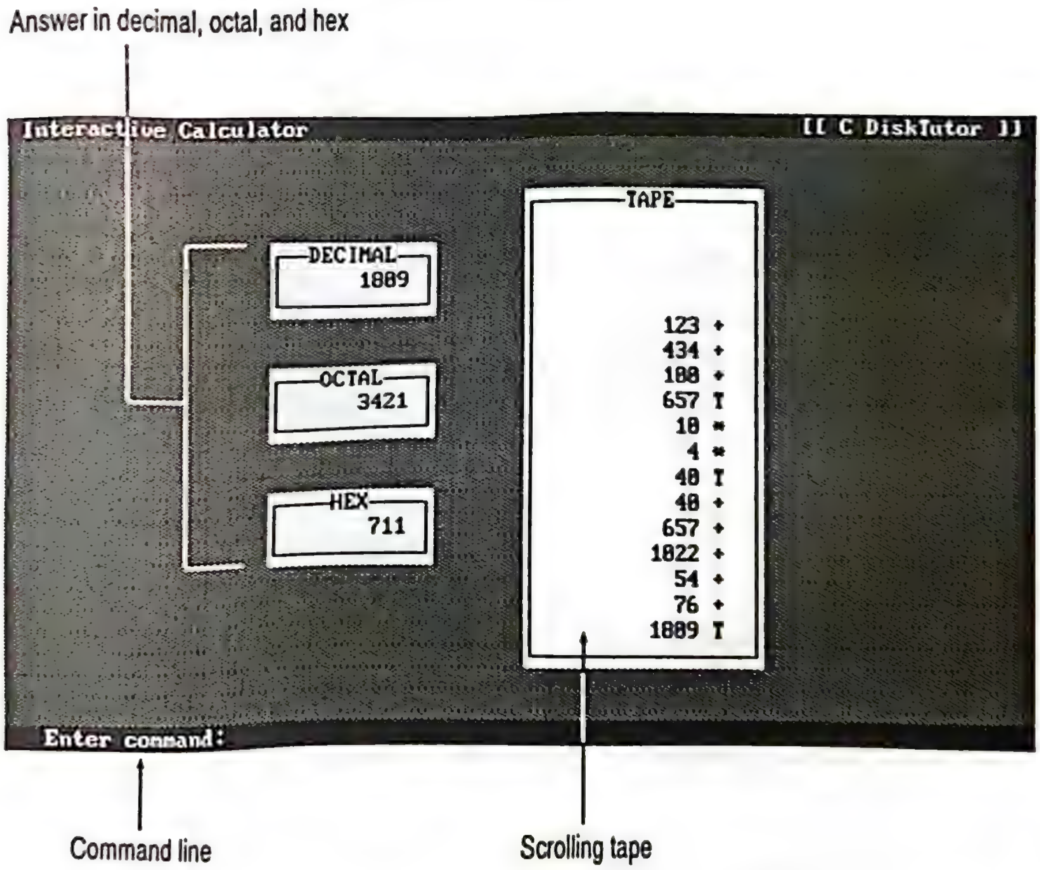


FIGURE 13-1
TUI Calculator screen

A skeleton program with the foregoing structure might look like this:

P13-1

```
/*
 * Case Study 1 (Skeleton Edition)
 *
 * Program:        CALC1.C       (a.k.a. P13-1.C)
 *
 * Author:         L. John Ribar
 *                 C DiskTutor
 *
 * Purpose:        This program implements a programmer's
 *                 calculator. It uses a TUI (Text-Based User
 *                 Interface) for interactive calculations,
 *                 and also allows execution using command-
 *                 line parameters.
 */

/*
   Preprocessor Directives
*/

#include <stdio.h>

/*
   Global and Static Variables
*/

/*
   Prototypes
*/

void doInteractive( void );

/*
   Main Processing Function
*/

void main( int argc, char *argv[] )
{
   int i;
   int total;

   if (argc < 2)  /* No parameters on the command line */
      doInteractive();
   else
```

```
    {
       for (i=1; i<argc; i++)
       {
          /* Process command-line parameters here. */
       }
       printf("result is %d decimal, 0%o octal, 0x%X hex,\n",
          total, total, total );
    }

}

void doInteractive( void )
{
    /* Do interactive mode-processing here. */
}
```

Naturally, if you run this program, nothing will happen, since the program as yet contains no processing code. Let's develop that next.

COMMAND-LINE PROCESSING

The first type of processing to be developed is the command-line interface. Assume that each command-line parameter will be either a number, or an operator, such as +, –, *, or /. Assume also that the program will start with the addition operation and continue adding until another operator is entered on the command line. In other words, the following command line:

```
calc 10 20 30
```

will print this result:

```
result is 60 decimal, 074 octal, and 0x3C hex
```

because the default operation is addition, and 10 + 20 + 30 gives a result of 60.

To make the code easier to read and understand, create an operator-mode variable, and #define statements to explain the current mode, like this:

```
/*
    Preprocessor Directives
*/
#define ADD      1
```

```
#define SUBTRACT 2
#define MULTIPLY 3
#define DIVIDE   4

/*
   Global Variables
*/
int command; /* Current command mode */
```

Finally, assign the default processing mode by using

```
command = ADD;
```

Now you can begin the processing of each parameter.

A simple way to handle the command-line parameters is to check the first character of each one. If the character is one of the operators, process the argument as a command. If not, process the argument as a number, and perform the appropriate operation, as follows:

```
total = 0;    /* This goes outside the for loop. */

switch (argv[i][0])
{
   case '+': command = ADD; break;
   case '-': command = SUBTRACT; break;
   case '*': command = MULTIPLY; break;
   case '/': command = DIVIDE; break;
   default:
      sscanf( argv[i], " %d", &thisNum );
      switch (command)
      {
         case ADD:      total += thisNum; break;
         case SUBTRACT: total -= thisNum; break;
         case MULTIPLY: total *= thisNum; break;
         case DIVIDE:   total /= thisNum; break;
      }
      break;
}
```

In the **switch** statement, you are using **argv[i][0]** to access the number **i** parameter, and the character at position 0 within that parameter.

This completes the operations needed for the command-line version of the calculator. The calculator program now looks like Program P13-2, and actually processes commands entered at the command line.

P13-2

```
/*
 * Case Study 1 (Command-line Version)
 *
 * Program:        CALC2.C      (a.k.a. P13-2.C)
 *
 * Author:         L. John Ribar
 *                 C DiskTutor
 *
 * Purpose:        This program implements a programmer's
 *                 calculator. It allows execution using
 *                 command-line parameters.
 */

/*
   Preprocessor Directives
*/

#include <stdio.h>

#define ADD      1
#define SUBTRACT 2
#define MULTIPLY 3
#define DIVIDE   4

/*
   Global and Static Variables
*/

/*
   Prototypes
*/

void doInteractive( void );

/*
   Main Processing Function
*/

void main( int argc, char *argv[] )
{
   int i;
   int total;
   int thisNum;
   int command;
```

```
      if (argc < 2)   /* No parameters on the command line */
         doInteractive();
      else
      {
         total = 0;          /* Clear the total. */
         command = ADD;      /* Default to addition. */

         for (i=1; i<argc; i++)   /* Process each command. */
         {
            switch (argv[i][0])
            {
               case '+': command = ADD; break;
               case '-': command = SUBTRACT; break;
               case '*': command = MULTIPLY; break;
               case '/': command = DIVIDE; break;
               default:
                  sscanf( argv[i], " %d", &thisNum );
                  switch (command)
                  {
                     case ADD:      total += thisNum; break;
                     case SUBTRACT: total -= thisNum; break;
                     case MULTIPLY: total *= thisNum; break;
                     case DIVIDE:   total /= thisNum; break;
                  }
                  break;
            }
         }
         printf("result is %d decimal, 0%o octal, 0x%X hex,\n",
            total, total, total );
      }

}

/* Local Functions */

void doInteractive( void )
{
   /* Interactive processing goes here. */
   printf("Interactive mode is not yet complete.\n");
}
```

INTERACTIVE PROCESSING

In interactive processing, the program reads commands entered by the user, one at a time, as would be done with a desk calculator. In order to

read numbers and commands from the user, the program needs an input routine; this input routine will accept keystrokes from the user until a nondigit command occurs.

Here is an implementation of this capability, using the **Inkey( )** function from the TUI Library:

```
/*
   Required header file for isdigit()
*/
#include <ctype.h>

/*
   Variables required
*/
int pos;
char tmp[20];

/*
   Processing starts here.
*/
pos = 0;     /* Start storing digits at the beginning of a
                string. */

ch = Inkey();
while ( isdigit(ch) )
{
   tmp[pos++] = ch;      /* Store the digit. */
   tmp[pos] = 0;         /* End the string with NULL char. */
   printf("%c", ch);     /* Display the keystroke. */
   ch = Inkey();         /* Read another key. */
}
printf("  %c\n", ch);   /* Display the command received. */
```

The foregoing segment of code reads keystrokes from the user until a nondigit character is pressed. The keystroke entered is checked with **isdigit( )**, which comes from the ctype.h header file. This **isdigit( )** function returns a True value if the character is a digit, or a False value if not. As each character is read, it is stored in a temporary string (**tmp**), which is later converted to a numeric value as follows:

```
/*
   Required header file for atoi()
*/
```

```
#include <stdlib.h>

aNum = atoi( tmp );
```

In the above code, **atoi( )** is a function that converts a string of digits into an integer.

Once the user's numbers and commands are received and read, the processing of numbers is similar to that done in the calculator's command-line version (see Program P13-2). The **doInteractive( )** function, without support for the TUI, would look like this:

```
void doInteractive( void )
{
   int pos;
   char tmp[20];
   char temp[80];
   char ch, lastCh;
   int command, total, aNum;

   /* Accept input; read pairs of (number plus command). */

   command = ADD;
   lastCh = '+';     /* Default to addition. */
   total = 0;

   for (;;)
   {
      printf("Enter command: ");
      /* Read digits, and then a command character. */
      ch = Inkey();
      if ( (ch == 'q')||(ch == 'Q') )    /* Q = Quit */
         break;
      if ( (ch == 'c')||(ch == 'C') )    /* C = Clear total */
      {
         total = 0;
         printf( "%12d C\n", total );
         continue;   /* Start next number */
      }
      if (ch == '=')                     /* '=' = Grand total */
      {
         printf( "%12d T\n", total );
         total = 0;
         continue;
      }
```

```
pos = 0;
while ( isdigit(ch) )
{
   tmp[pos++] = ch;
   tmp[pos] = 0;
   Say( 24, 18, tmp );
   ch = Inkey();
}
if (ch == 13)      /* Carriage return */
   ch = lastCh;    /* Default to last operation. */
aNum = atoi( tmp );
printf( "%12d %c\n", aNum, ch );

switch (command)
{
   case ADD:
      total += aNum;   break;
   case SUBTRACT:
      total -= aNum;   break;
   case MULTIPLY:
      total *= aNum;   break;
   case DIVIDE:
      total /= aNum;   break;
}

/* Print results. */
printf( tmp, "SubTotal: %10d  0%o  0x%X\n", total,
       total, total );

/* Get next command. */
switch (ch)
{
   case '+':
      command = ADD;
      break;
   case '-':
      command = SUBTRACT;
      break;
   case '*':
      command = MULTIPLY;
      break;
   case '/':
      command = DIVIDE;
      break;
```

```
            case '=':
               printf( "%12d T\n", total );
               /* Drop through to clear the total. */
            case 'c':
            case 'C':
               total = 0;
               break;
            default:
               break;
         }
         if ((ch == 'q')||(ch == 'Q'))
            break;
         lastCh = ch;
      }

      printf( "Press any key to continue ...");
      Inkey();

      /* Clear up the screen. */
      clrscr();
   }
```

ADDING THE TUI INTERFACE

Adding the interface involves the incorporation of screen controls, and a different display method for all data to be printed on the screen. To display the initial screen (Figure 13-1), the following code is used:

```
/* Set up the TUI. */
clrscr();
Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
SayTitle( "Interactive Calculator" );

/* Set up the window for calculator tape. */
DrawBox( 3, 40, 21, 58, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* tape */

/* Set up the windows for different values. */
DrawBox( 5, 20, 7, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* decimal */
DrawBox(10, 20, 12, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* octal */
DrawBox(15, 20, 17, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* hex */
```

```
/* Draw titles on the windows. */
Say( 3, 48, "TAPE" );
Say( 5, 23, "DECIMAL");
Say(10, 24, "OCTAL" );
Say(15, 25, "HEX");
```

Next, all the **printf( )** statements used to display data or informational messages are changed so they can be displayed on the help (bottom) line using **SayHelp( )**, or in the Decimal, Octal, and Hex windows. Since most of these messages include parameters, use **sprintf( )** to build the message as a character string, and then use **Say( )** or **SayHelp( )** with the string.

For example, to display the last number and command entered, you might use the following code. It builds a message using **sprintf( )**, and then displays the message in the tape window using **Say( )**.

```
/*
   Put this declaration in the global area.
*/
char outLine[80];

/*
   Then use this type of code.
*/
sprintf( outLine, "%12d %c", aNum, ch );
Say( 20, 42, outLine );
```

In addition, as numbers are placed at the bottom of the tape window, you will want to scroll the old information up. To do this, use a call to **Scroll( )**, as follows. Remember—ANSI terminals will not support the scroll functions!

```
Scroll( 4, 41, 20, 57, -1, NORMAL );
```

ONE COMPLETE SOLUTION

Here is one complete solution for the Programmer's Calculator—Program P13-3, which uses the design criteria developed within this chapter.

P13-3

```
/*
 * Case Study 1
 *
 * Program:        CALC.C       (a.k.a. P13-3.C)
 *
 * Author:         L. John Ribar
 *                 C DiskTutor
 *
 * Purpose:        This program implements a programmer's
 *                 calculator. It uses a TUI (Text-based User
 *                 Interface) for interactive calculations,
 *                 and also allows execution using command-
 *                 line parameters.
 *
 * Build Sequence:
 *                 WCL P13-3 SCREEN
 *              or WCL CALC SCREEN
 */

/*
   Preprocessor Directives
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "screen.h"

#define ADD      1
#define SUBTRACT 2
#define MULTIPLY 3
#define DIVIDE   4

/*
   Global and Static Variables
*/

/*
   Prototypes
*/

void doInteractive( void );

/*
   Main Processing Function
```

```
 */

void main( int argc, char *argv[] )
{
    int position;
    int i;
    int total;
    int thisNum;
    int command;

    if (argc < 2)  /* No parameters on the command line */
       doInteractive();
    else
    {
       total = 0;
       command = ADD;

       for (i=1; i<argc; i++)
       {
          switch (argv[i][0])
          {
             case '+': command = ADD; break;
             case '-': command = SUBTRACT; break;
             case '*': command = MULTIPLY; break;
             case '/': command = DIVIDE; break;
             default:
                sscanf( argv[i], " %d", &thisNum );
                switch (command)
                {
                   case ADD:      total += thisNum; break;
                   case SUBTRACT: total -= thisNum; break;
                   case MULTIPLY: total *= thisNum; break;
                   case DIVIDE:   total /= thisNum; break;
                }
                break;
          }
       }
       printf("result is %d decimal, 0%o octal, 0x%X hex,\n",
          total, total, total );
    }

}

/* Local Functions */
```

```
void doInteractive( void )
{
   int pos;
   char tmp[20];
   char temp[80];
   char ch, lastCh;
   int command, total, aNum;

   /* Set up the TUI. */
   clrscr();
   Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
   SayTitle( "Interactive Calculator" );

   /* Set up the window for calculator tape. */
   DrawBox( 3, 40, 21, 58, VERT, HORIZ, NORMAL, FILL, SHADOW );
   /* tape */

   /* Set up the windows for different values. */
   DrawBox( 5, 20, 7, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
   /* decimal */
   DrawBox(10, 20, 12, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
   /* octal */
   DrawBox(15, 20, 17, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
   /* hex */

   /* Draw titles on the windows. */
   Say( 3, 48, "TAPE" );
   Say( 5, 23, "DECIMAL");
   Say(10, 24, "OCTAL" );
   Say(15, 25, "HEX");

   /* Now accept input; read pairs of number plus command. */
   command = ADD;
   lastCh = '+';    /* Default first operation to addition */
   total = 0;

   for (;;)
   {
      SayHelp("Enter command: ");
      /* Read digits, and then a command character. */
      ch = Inkey();
      if ( (ch == 'q')||(ch == 'Q') )
         break;
      if ( (ch == 'c')||(ch == 'C') )
      {
```

```
        total = 0;
        sprintf( temp, "%12d C", total );
        Scroll( 4, 41, 20, 57, -1, NORMAL );
        Say( 20, 42, temp );
        continue;
    }
    if (ch == '=')
    {
        sprintf( temp, "%12d T", total );
        Scroll( 4, 41, 20, 57, -1, NORMAL );
        Say( 20, 42, temp );

        /* Now reset for the next operation. */
        total = 0;
        lastCh = '+';
        command = ADD;
        continue;
    }
    pos = 0;
    while ( isdigit(ch) )
    {
        tmp[pos++] = ch;
        tmp[pos] = 0;
        Say( 24, 18, tmp );
        ch = Inkey();
    }
    if (ch == 13)  /* carriage return */
        ch = lastCh;
    aNum = atoi( tmp );
    sprintf( temp, "%12d %c", aNum, ch );
    Scroll( 4, 41, 20, 57, -1, NORMAL );
    Say( 20, 42, temp );

    switch (command)
    {
        case ADD:
            total += aNum;   break;
        case SUBTRACT:
            total -= aNum;   break;
        case MULTIPLY:
            total *= aNum;   break;
        case DIVIDE:
            total /= aNum;   break;
   }
```

```
        /* Print results */
        sprintf( tmp, "%10d", total );
        Say( 6, 21, tmp );
        sprintf( tmp, "%10o", total );
        Say(11, 21, tmp );
        sprintf( tmp, "%10X", total );
        Say(16, 21, tmp );

        /* Get next command. */
        switch (ch)
        {
           case '+':
              command = ADD;
              break;
           case '-':
              command = SUBTRACT;
              break;
           case '*':
              command = MULTIPLY;
              break;
           case '/':
              command = DIVIDE;
              break;
           case '=':
              sprintf( temp, "%12d T", total );
              Scroll( 4, 41, 20, 57, -1, NORMAL );
              Say( 20, 42, temp );
           case 'c':
           case 'C':
              total = 0;
              break;
           default:
              break;
        }
        if ((ch == 'q')||(ch == 'Q'))
           break;
        lastCh = ch;
     }

     SayHelp( "Press any key to continue ...");
     Inkey();

     /* Clear up the screen. */
     clrscr();
}
```

ENHANCING THE SOLUTION

Two simple changes to Program P13-3 will give it additional capabilities, as suggested in the optional requirements for this project.

PRINTING CAPABILITY

To add a simultaneous printing facility to the calculator program, change each **printf( )** statement to a **printf( )** with an **fprintf( )**. For instance, change

```
printf("%12d %c\n", aNum, ch);
```

to read as follows:

```
printf("12d %c\n", aNum, ch);
fprintf( stdprn, "%12d %c\n", aNum, ch);
```

ADDING A MEMORY

Adding a memory capability to your calculator is as simple as adding two commands: one to place the current total in memory, and one to recall the total as a number for the current calculation. With the memory function added, your **doInteractive( )** function might look like this:

```
void doInteractive( void )
{
   int pos;
   char tmp[20];
   char temp[80];
   char ch, lastCh;
   int command, total, aNum;
   int memory;

   /* Set up the TUI. */
   clrscr();
   Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
   SayTitle( "Interactive Calculator" );

   /* Set up the window for calculator tape. */
   DrawBox( 3, 40, 21, 58, VERT, HORIZ, NORMAL, FILL, SHADOW );
   /* tape */
```

```
/* Set up the windows for different values. */
DrawBox( 5, 20, 7, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* decimal */
DrawBox(10, 20, 12, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* octal */
DrawBox(15, 20, 17, 32, VERT, HORIZ, NORMAL, FILL, SHADOW );
/* hex */

/* Draw titles on the windows. */
Say( 3, 48, "TAPE" );
Say( 5, 23, "DECIMAL");
Say(10, 24, "OCTAL" );
Say(15, 25, "HEX");

/* Now accept input; read pairs of number plus command. */
command = ADD;
lastCh = '+';    /* Default first operation to addition. */
total = 0;
memory = 0;

for (;;)
{
   SayHelp("Enter command: ");
   /* Read digits, and then a command character. */
   ch = Inkey();
   if ( (ch == 'q')||(ch == 'Q') )
      break;
   if ( (ch == 'c')||(ch == 'C') )
   {
      total = 0;
      sprintf( temp, "%12d C", total );
      Scroll( 4, 41, 20, 57, -1, NORMAL );
      Say( 20, 42, temp );
      continue;
   }
   if (ch == '=')
   {
      sprintf( temp, "%12d T", total );
      Scroll( 4, 41, 20, 57, -1, NORMAL );
      Say( 20, 42, temp );
      total = 0;
      continue;
   }
   pos = 0;
   while ( isdigit(ch) )
```

```
{
   tmp[pos++] = ch;
   tmp[pos] = 0;
   Say( 24, 18, tmp );
   ch = Inkey();
}
if (ch == 13)  /* carriage return */
   ch = lastCh;
if ( (ch == 'R')||(ch == 'r') )
{
   ch = lastCh;
   aNum = memory;
}
else
   aNum = atoi( tmp );
sprintf( temp, "%12d %c", aNum, ch );
Scroll( 4, 41, 20, 57, -1, NORMAL );
Say( 20, 42, temp );

switch (command)
{
   case ADD:
      total += aNum;   break;
   case SUBTRACT:
      total -= aNum;   break;
   case MULTIPLY:
      total *= aNum;   break;
   case DIVIDE:
      total /= aNum;   break;
}

/* Print results. */
sprintf( tmp, "%10d", total );
Say( 6, 21, tmp );
sprintf( tmp, "%10o", total );
Say(11, 21, tmp );
sprintf( tmp, "%10X", total );
Say(16, 21, tmp );

/* Get next command. */
switch (ch)
{
   case '+':
      command = ADD;
      break;
```

```
            case '-':
               command = SUBTRACT;
               break;
            case '*':
               command = MULTIPLY;
               break;
            case '/':
               command = DIVIDE;
               break;
            case 'M':    /* Put total into memory. */
               sprintf( temp, "%12d M", total );
               Scroll( 4, 41, 20, 57, -1, NORMAL );
               Say( 20, 42, temp );
               memory = total;
               break;
            case '=':
               sprintf( temp, "%12d T", total );
               Scroll( 4, 41, 20, 57, -1, NORMAL );
               Say( 20, 42, temp );
            case 'c':
            case 'C':
               total = 0;
               break;
            default:
               break;
         }
         if ((ch == 'q')||(ch == 'Q'))
            break;
         lastCh = ch;
      }

      SayHelp( "Press any key to continue ...");
      Inkey();

      /* Clear up the screen. */
      clrscr();
   }
```

C H A P T E R 14

CASE STUDY 2:
A FILE-DUMP UTILITY

The requirements for Case Study 2, the File-dump Utility, were presented in Chapter 10, and are listed again here. In this chapter, you will see the program developed step by step, with additional functions added in small increments. The File-dump Utility will be able to:

- Send output to the screen, the printer, or a disk file.
- Allow the user to specify multiple filenames and commands on the command line.
- Toggle between ASCII and hexadecimal output modes for each file, based on command-line parameters. The ASCII output mode should look similar to Figure 14-1.
- If the file is dumped to the screen, allow the user to stop the display after each screenful of data, in order to read what is being displayed.

The following optional assignments for the file-dump case study were also presented in Chapter 10. These optional requirements are intended to help you start thinking about your own ideas for enhancements.

- Allow the user to move forward and backward through the file when viewing it on the screen.
- Use the Screen and TUI Library from Chapter 8 to create an interactive environment, allowing selection of filenames from within the program, display of the files within a box on the screen (a window), and use of special keys (function keys, arrows, and so on) for commands.

Certainly the purpose of this case study is to practice programming in C, but it's also intended to help you realize that you can now write your own useful programs in the C language.

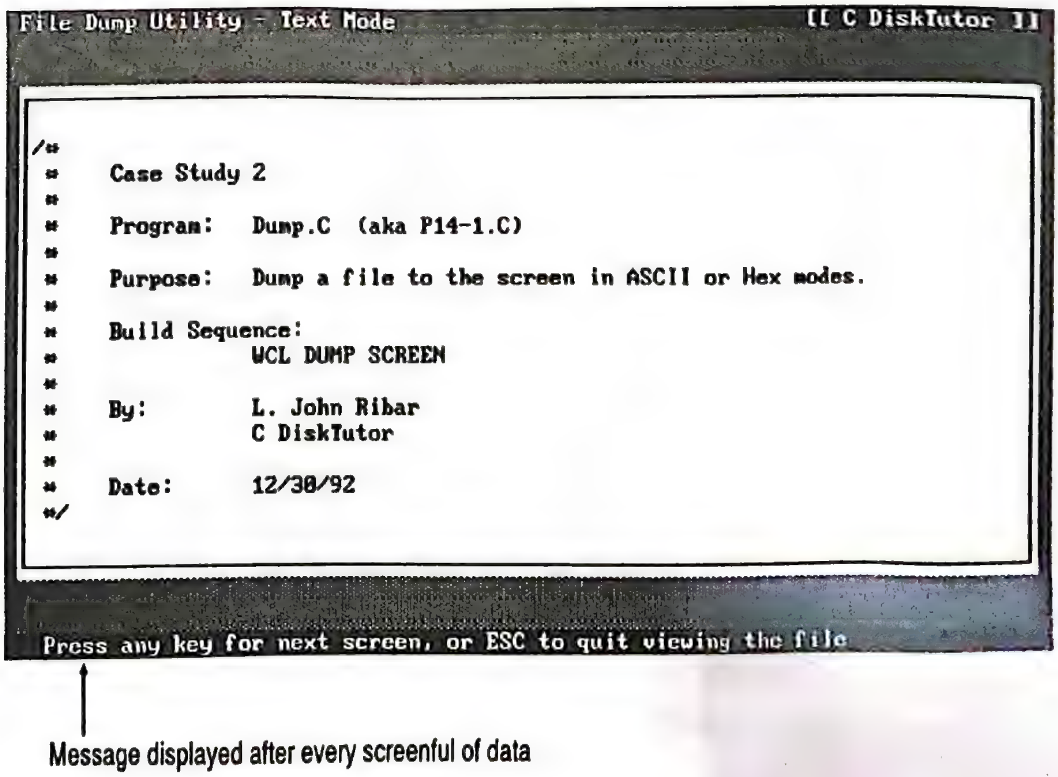


FIGURE 14-1
A file dumped to the screen in ASCII text format

STARTING THE PROJECT

Since this project requires that multiple commands and filenames be specified by the user on the command line, the file-dump program will be designed as a big loop. Within the loop, the program will read and act upon each command parameter, one at a time. Pseudocode for the program might look like this:

```
Set up Defaults
FOR Each Command Line Parameter DO
   IF the Parameter Starts with '/' THEN
      Process the Command
   ELSE
      Dump the File Specified by the Current Parameter
   END IF
END FOR loop
```

Of course, the processing of commands and dumping of the file will take more than a few lines of code. To get started, consider the skeleton program that follows (Program P14-1), which is written with the above pseudocode in mind. The defaults for the program are implemented as global variables located at the top of the file. Preprocessor **#define** statements allow a more readable program, by replacing numbers with meaningful text.

P14-1

```
/*
 *    Case Study 2 (Skeleton Version)
 *
 *    Program:   DUMP1.C  (a.k.a. P14-1.C)
 *
 *    Purpose:   Dump a file to the screen, printer, or file
 *               in ASCII or hexadecimal format.
 *
 *    Build Sequence:
 *               WCL DUMP1 SCREEN  or
 *               WCL P14-1 SCREEN
 *
 *    By:        L. John Ribar
 *               C DiskTutor
 *
 *    Date:      12/30/92
 */
```

```
#include <stdio.h>

/*
   Preprocessor Definitions
*/

/* Printing Mode */
#define modeASCII 0
#define modeHEX   1

/* Printing Location */
#define outSCREEN  0
#define outPRINTER 1
#define outFILE    2

/*
   Global and Static Variables
*/

int hexMode = modeASCII;/*
                            Printing in hex mode?
                            0 = ASCII mode
                            1 = Hex mode
                         */
int outMode = outSCREEN;/*
                            Where is the output going?
                            0 = SCREEN
                            1 = PRINTER
                            2 = FILE
                         */


void main( int argc, char *argv[] )
{
   char inFile[80];        /* Current file being read */
   int i;                  /* Indexing variable */

   if (argc < 2)           /* Provide help */
   {
      printf("Usage: dump [/h] [/p] file [file ...]\n");
      return;
   }

   inFile[0] = 0;          /* No file is open yet */
```

```
    for (i=1; i<argc; i++)
    {
        if (argv[i][0] == '/')  /* Command-line parameters */
        {
            /*** Process the commands ***/
        }
        else
        {
            /*** Dump the file ***/
        }
    }
}
```

COMMAND PROCESSING

To process the commands entered by the user, the program uses a switch statement; and since the first character is the slash character, the switch statement looks at the second character of each command. The switch statement might look like this:

```
switch ( tolower(argv[i][1]) )   /* Convert to lowercase */
{
    case 'h':   /* Hex output */
        break;
    case 'a':   /* ASCII output */
        break;
    case 'p':   /* Printer output */
        break;
    case 's':   /* Screen output */
        break;
    default:
        printf( "ERROR: %s is not a valid command.\n", argv[i] );
        break;
}
```

Notice that the **tolower()** function used in the **switch** statement ensures that the command is a lowercase letter. This function is found in the ctype.h header file, and lets you check for only the lowercase letters in your **case** statements. Notice, also, that an error message is provided as the default case, letting the user know a bad command was entered.

Because this program has several commands available for the user, a help screen is provided when the program is started without any parameters. The help information will look something like the following:

```
Usage: dump [command(s)] file(s)
Valid Commands are:
     /h     Use Hex Mode output
     /a     Use ASCII Mode output
     /p     Send output to printer
     /fx    Send output to file named x
     /s     Send output to screen
```

To start filling in the case statements, you need to determine what is to be done in each case. For instance, the **h** and **a** commands need to set the default processing mode. The code to perform this function is quite simple:

```
switch ( tolower(argv[i][1]) )    /* Convert to lowercase
*/
{
   case 'h':   /* Hex output command */
      hexMode = modeHEX;
      break;
   case 'a':   /* ASCII output command */
      hexMode = modeASCII;
      break;
   case 'p':   /* Printer output command */
      break;
   case 's':   /* Screen output command */
      break;
   default:
      printf( "ERROR: %s is not a valid command.\n", argv[i] );
      break;
}
```

Very little processing is required for these commands

Before setting the output destination (using the **p**, **f**, and **s** commands), you need to make sure that files opened by previous commands are not still open (remember, the program will be handling multiple files). This is simple to check; if the current output file is **stdprn** or **stdout**, no files need to be closed; otherwise, you need to close **outFile**. A statement to perform these functions looks like this:

```
if ((outFile!=stdout)&&(outFile!=stdprn))
   fclose(outFile);
```

Changing the output destination also involves changing the **outMode** flag, and if output is being sent to a file, the output file must be opened.

When all these operations are in place, the command processing section looks like this:

```
/*
   Add these two definitions to the global variables.
*/

FILE *outFile = stdout; /*
                           Output file
                        */
char inName[80];        /*
                           Input filename
                        */
char outName[80];       /*
                           Output filename, for page header
                        */

switch ( tolower(argv[i][1]) )   /* Convert to lowercase */
{
   case 'h':   /* Hex output command */
      hexMode = modeHEX;
      break;
   case 'a':   /* ASCII output command */
      hexMode = modeASCII;
      break;
   case 'p':   /* Printer output command  */
      outMode = outPRINTER;
      /* First, make sure to close any previously opened
         file. */
      if ((outFile!=stdout)&&(outFile!=stdprn))
         fclose(outFile);
      outFile = stdprn;
      break;
   case 'f':   /* File output command */
      outMode = outFILE;
      /* First, make sure to close any previously opened
         file. */
      if ((outFile!=stdout)&&(outFile!=stdprn))
         fclose(outFile);
      /*
         The file command looks like /fFILENAME, so start
         reading the filename from position 2 of the command.
      */
      outFile = fopen( &argv[i][2], "w" );
      if (outFile == NULL)
```

```
            {
               printf( "ERROR: %s cannot be opened for output.\n.",
                  &argv[i][2] );
               /* If there is an error opening the file, then
                  default back to using the screen, so that program
                  can continue. */
               outFile = stdout;
               outMode = outSCREEN;
            }
            else
               strcpy( outName, &argv[i][2] );
            break;
         case 's':   /* Screen output command */
            outMode = outSCREEN;
            /* First, make sure to close any previously opened
               file. */
            if ((outFile!=stdout)&&(outFile!=stdprn))
               fclose(outFile);
            outFile = stdout;
            break;
         default:
            printf( "ERROR: %s is not a valid command.\n", argv[i] );
            break;
}
```

This completes the command processing necessary for this program.

DUMPING THE FILES

The command processing loop developed in the last section is invoked whenever the first character of a command-line parameter is a slash character. Now you will add processing for the other possible parameters, assumed to be names of files to be dumped.

First, you need to add an **else** clause to the **if** statement developed for the command processing. If the entered command does not start with a slash character, you need to save the parameter as a filename, and attempt to dump the file, like this:

```
inFile[0] = 0;    /* Set the filename empty to start. */

for (i=1; i<argc; i++)
{
   if (argv[i][0] == '/')  /* Command-line parameters */
   {
```

```
        /*** Process the commands ***/
    }
    else
        strcpy( inFile, argv[i] );

    /*
        If this parameter wasn't a command, we should have a file
        name to print!
    */
    if (inFile[0] != 0)
    {
        dumpFile(inFile);       /* Dump this file. */
        inFile[0] = 0;          /* Reset for next file. */
    }

}
```

This is all that is required by the **main()** processing section of the program. Now you need to develop the **dumpFile()** function to send the file to its destination, based on the selections made through the command line.

The dumpFile() Function

The purpose of the **dumpFile()** function is to open and print the requested file to the requested location, using either hexadecimal (hex) or ASCII output format. This operation will take the following steps:

1. Try to open the file. If hex output was selected, open the file as a binary file. Otherwise, open it in standard ASCII text (translated) mode.
2. If the file cannot be opened, display an error message, and return to the main program.
3. If hex mode was selected, read 16 bytes from the file. Otherwise, read one line from the file.
4. Display or print the line in the chosen format.
5. Check how many lines have been printed. The screen output handles 20 lines between pauses for user input to continue. The printer and file output destinations print a header for each page, 55 lines of data per page, and a formfeed character at the end of the page.

```
                printf("\n");

                if (ch == Esc)
                   break;       /* Finished! */
                else
                   currentRow = 1;
             }
          } while (!feof(inF) );
       }
       printf( "Press any key to continue ...");

       /* Leave the file displayed until a key is pressed. */
       ch = Inkey();
       printf("\n\n");
    }
    else if ( (outMode == outPRINTER)||(outMode == outFILE) )
    {
       if (outMode == outPRINTER)
          printf( "Dumping file %s to the printer ... \n",
          fName );
       else
          printf( "Dumping file %s to file %s ... \n", fName,
          outName );

       /* Print the file. */
       printHeader();
       currentRow = 0;
       curLoc = 0L;

       if (hexMode)
       {
          do
          {
             strcpy( inLine, "................" );

             /* First clear all values to zero. */
             for (i=0; i<16; i++)
                inLine[i+17] = 0;

                /* Now read in the characters from the file. */
             for (i=0; i<16; i++)
             {
                inLine[i+17] = getc( inF );
                if (feof(inF))
                   break;
```

```
      }

      /* Now print the results. */
      fprintf( outFile, "[%06X]", curLoc );
      for (i=0; i<16; i++)
      {
         if ( (i%4) == 0)
            fprintf( outFile, "   ");
         fprintf( outFile, "%02x", inLine[i+17]);
         if (inLine[i+17] >= ' ')
            inLine[i] = inLine[i+17];
      }
      fprintf( outFile, "   [%s]\n", inLine );

      currentRow++;
      if (currentRow > 55)
      {
         fprintf( outFile, "%c", 12 ); /* Formfeed */
         printHeader();
         currentRow = 0;
      }
      curLoc += 16;  /* Reading 16 bytes at a time */
   } while (!feof(inF) );
   fprintf( outFile, "%c", 12 ); /* Final formfeed */
}
else
{
   do
   {
      fgets( inLine, 128, inF );
      inLine[ strlen(inLine)-1 ] = 0;  /* Remove CR/LF */

      fprintf( outFile, "%s\n", inLine );
      currentRow++;
      if (currentRow > 55)
      {
         fprintf( outFile, "%c", 12 ); /* Formfeed */
         printHeader();
         currentRow = 0;
      }
   } while (!feof(inF) );
   fprintf( outFile, "%c", 12 ); /* Final formfeed */
}
printf( "Press any key to continue ...");
Inkey();
```

```
        printf("\n\n");
    }
}
```

About the Hexadecimal Output Format The hexadecimal mode display is constructed in the following manner: sixteen 2-digit hex values are displayed, followed by 16 character values. The following steps are used to create this display. First, 16 bytes are read from the input file and stored in **inLine**, in positions 17 through 33. These positions are used so that positions 0 through 15 can be used as a character representation, of each byte, to be displayed at the end of each output line. For each of these 16 bytes just read, a 2-character hexadecimal version of the number is printed, using the **%x** specifier in **printf( )**. After all 16 hexadecimal sets are printed, the ASCII representation (stored in characters 0 through 15 of **inLine**) is printed, and the program moves to the next 16 bytes in the file.

The printHeader( ) Function

The **printHeader( )** function simply prints a header line at the top of each page sent to the printer or to a disk file. The header line includes the name of the file being printed, and the output format (hexadecimal or ASCII). The **printHeader( )** function centers this information on the top line of the page by printing enough spaces to place the text in the correct position. The code looks like this:

```
void printHeader( void )
{
    int i, l;
    sprintf( outLine, "---------%s Dump of File %s ---------",
        (hexMode == modeHEX) ? "-- Hex" : " ASCII", inName );
    l = strlen( outLine );
    l = (80-l)/2;   /* Center the header. Find out how many
                        spaces we need. */
    for (i=0; i<l; i++)
        fprintf( outFile, " " ); /* Print the spaces */
    fprintf( outFile, "%s\n\n", outLine );
}
```

ADDING A TUI FRONT-END

To add a Text-based User Interface as the front-end to the file-dump program, you first need to set up the TUI screen, like this:

```
/* First, clear the screen. */
clrscr();

/* Put up our TUI. */
Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
SayTitle( "File Dump Utility" );
```

To make the program look smarter, you might change the contents of the title bar based on the output mode, like this:

```
if (hexMode == modeHEX)
   SayTitle( "File Dump Utility - Hex Mode" );
else
   SayTitle( "File Dump Utility - Text Mode" );
```

This is especially useful when the dump is being sent to the screen. Remember that as commands are processed, the title should change if the output mode changes.

Next, all the **printf( )** statements used to display error or informational messages are changed so they can be displayed on the help line, using **SayHelp( )**. Since most of these messages include parameters, use **sprintf( )** to build each message as a character string, and then use **SayHelp( )** with the string.

As an example, this code will inform the user when a file cannot be opened:

```
/*
   Put this declaration in the global area!
*/
char outLine[80];


sprintf( outLine,
   "File %s cannot be opened. Press any key ...", fName );
SayHelp( outLine );
Inkey();  /* Wait for a keystroke. */
ClearHelp();
```

This segment of code builds a message using **sprintf( )**, displays the message using **SayHelp( )**, waits for the user to press a key using **Inkey( )**, and then erases the message using **ClearHelp( )**. While the program waits for the next keystroke, the user has the opportunity to read the message before anything else happens in the program.

Finally, the display of the file itself will occur within a window on the screen. Display the window using code something like this:

```
/* Draw the display box. */
DrawBox( 3, 1, 21, 78, VERT, HORIZ, NORMAL, FILL, SHADOW );
```

This sets up an area on the screen where you can place the output from the **dumpFile( )** function. The lines are displayed using **Say( )**, rather than **printf( )** statements, to position each line in a specific location.

ONE COMPLETE SOLUTION

Following is Program P14-2, a complete solution to Case Study 2, as designed in the foregoing sections and using the TUI from Chapter 8.

P14-2

```
/*
 *    Case Study 2
 *
 *    Program:    DUMP.C  (a.k.a. P14-1.C)
 *
 *    Purpose:    Dump a file to the screen in ASCII or
 *                hexadecimal format
 *
 *    Build Sequence:
 *                WCL DUMP SCREEN
 *             or WCL P14-2 SCREEN
 *
 *    By:         L. John Ribar
 *                C DiskTutor
 *
 *    Date:       12/30/92
 */

/*
   Preprocessor Directives
*/

#include <stdio.h>
#include <ctype.h>
#include "screen.h"
```

```
/* Printing mode */
#define modeASCII 0
#define modeHEX   1

/* Printing location */
#define outSCREEN  0
#define outPRINTER 1
#define outFILE    2

/*
   Global and Static Variables
*/

int hexMode = modeASCII;/*
                          Printing in hex mode?
                          0 = ASCII mode
                          1 = Hex mode
                          (see #define's above)
                        */
int outMode = outSCREEN;/*
                          Where is the output going?
                          0 = SCREEN
                          1 = PRINTER
                          2 = FILE
                          (see #define's above)
                        */
FILE *outFile = stdout; /*
                          Output file
                        */
char inName[80];        /*
                          Input filename
                        */
char outLine[80];       /*
                          Used for help messages
                        */
char outName[80];       /*
                          Output filename
                        */
/*
   Local Prototypes
*/

void dumpFile( char *fName );
void BaseHelp( void );
```

```
void printHeader( void );

/*
   Main Processing Area
*/
void main( int argc, char *argv[] )
{
   char inFile[80];       /* Current file being read */
   char tmp[10];          /* Conversion area */
   int i;                 /* Indexing variable */

   if (argc < 2)
   {
      printf("Usage: dump [command(s)] file(s)\n");
      printf("Valid Commands are:\n");
      printf("    /h    Use Hexadecimal Mode output\n");
      printf("    /a    Use ASCII Mode output\n");
      printf("    /p    Send output to printer\n");
      printf("    /fx   Send output to file named x\n");
      printf("    /s    Send output to screen\n");
      return;
   }

   inFile[0] = 0;         /* No file is open yet. */
   outName[0] = 0;        /* No output file selected. */

   /* First, clear the screen. */
   clrscr();

   /* Put up our TUI. */
   Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
   if (hexMode)
      SayTitle( "File Dump Utility - Hex Mode" );
   else
      SayTitle( "File Dump Utility - Text Mode" );

   /*
      Process each command-line parameter. Each parameter
      can be either an actual command (beginning with a
      slash character), or a filename.
   */
   for (i=1; i<argc; i++)
   {
      if (argv[i][0] == '/')  /* Command-line parameters */
      {
```

```
switch ( tolower(argv[i][1]) )    /* Convert to
                                     lowercase */
{
   case 'h':   /* Hex output command */
      hexMode = modeHEX;
      SayTitle( "File Dump Utility - Hex Mode" );
      break;
   case 'a':   /* ASCII output command */
      hexMode = modeASCII;
      SayTitle( "File Dump Utility - Text Mode" );
      break;
   case 'p':   /* Printer output command */
      outMode = outPRINTER;
      /* First, make sure to close any previously
         opened file.*/
      if ((outFile!=stdout)&&(outFile!=stdprn))
         fclose(outFile);
      outFile = stdprn;
      break;
   case 'f':   /* File output command */
      outMode = outFILE;
      /* First, make sure to close any previously
         opened file.*/
      if ((outFile!=stdout)&&(outFile!=stdprn))
         fclose(outFile);
      outFile = fopen( &argv[i][2], "w" );
      if (outFile == NULL)
      {
         sprintf( outLine,
            "%s cannot be opened for output. Press
            any key ...", &argv[i][2] );
         SayHelp( outLine );
         Inkey();  /* Wait for a keystroke. */
         ClearHelp();
         outFile = stdout;
         outMode = outSCREEN;
      }
      else
         strcpy( outName, &argv[i][2] );
      break;
   case 's':   /* Screen output command */
      outMode = outSCREEN;
      /* First, make sure to close any previously
         opened file.*/
      if ((outFile!=stdout)&&(outFile!=stdprn))
```

```
                    fclose(outFile);
                 outFile = stdout;
                 break;
              default:
                 sprintf( outLine,
                    "%s is not a valid command. Press any
                    key ...", argv[i] );
                 SayHelp( outLine );
                 Inkey();
                 ClearHelp();
                 break;
           }
        }
        else
           strcpy( inFile, argv[i] );

        if (inFile[0] != 0)
        {
           strcpy(inName, inFile); /* For headers */
           dumpFile(inFile);    /* Dump this file */
           inFile[0] = 0;       /* Reset for next file */
        }
     }

     /* Clean up the screen.*/
     clrscr();
  }


  void dumpFile( char *fName )
  {
     char inLine[128];    /* Last line that was read */
     int currentRow;      /* Current row on the screen */
     unsigned long curLoc;/* Current location in the file */
     int i;               /* Indexing variable */
     char ch;             /* Character input variable */
     char tmp[10];        /* Conversion area */
     FILE *inF;           /* File being read */

     if (hexMode == modeHEX)
        inF = fopen( fName, "rb" );
     else
        inF = fopen( fName, "r" );

     if (inF == NULL)
```

```
{
   sprintf( outLine, "Error! Could not open file %s.
            Press any key ...",
      fName );
   SayHelp( outLine );
   Inkey();        /* Wait for a keypress. */
   BaseHelp();
   return;
}

if (outMode == outSCREEN)
{
   BaseHelp();

   /* Draw the display boxes. */
   DrawBox( 3, 1, 21, 78, VERT, HORIZ, NORMAL, FILL,
            SHADOW );
   currentRow = 4;

   /* Display the file. */
   if (hexMode)
   {
      do
      {
         strcpy( inLine, "................" );
         for (i=0; i<16; i++)
            inLine[i+17] = 0;

         for (i=0; i<16; i++)
         {
            inLine[i+17] = getc( inF );
            if (feof(inF))
               break;
         }

         for (i=0; i<16; i++)
         {
            sprintf( tmp, "%02x", inLine[i+17]);
            Say( currentRow, 5+(i*3)+(i/8)*3, tmp );
            if (inLine[i+17] >= ' ')
               inLine[i] = inLine[i+17];
         }
         Say( currentRow, 5+(16*3)+6, inLine );

         currentRow++;
```

```
            if (currentRow > 20)
            {
                ch = Inkey();
                if (ch == Esc)
                {
                    break;      /* Finished! */
                }
                else
                {
                    Scroll( 4, 2, 20, 77, 0, NORMAL );
                    currentRow = 4;
                }
            }
        } while (!feof(inF) );
    }
    else
    {
        do
        {
            fgets( inLine, 128, inF );
            inLine[ strlen(inLine)-1 ] = 0;  /* Remove CR/LF */
            inLine[74] = 0;   /* In case line is too big for
                                 the window. */

            Say( currentRow, 2, inLine );
            currentRow++;
            if (currentRow > 20)
            {
                ch = Inkey();
                if (ch == Esc)
                {
                    break;      /* Finished! */
                }
                else
                {
                    Scroll( 4, 2, 20, 77, 0, NORMAL );
                    currentRow = 4;
                }
            }
        } while (!feof(inF) );
    }
    SayHelp( "Press any key to continue ...");

    /* Leave the file displayed until a key is pressed. */
    ch = Inkey();
```

```
    ClearHelp();
}
else if ( (outMode == outPRINTER)||(outMode == outFILE) )
{
    if (outMode == outPRINTER)
        sprintf( outLine, "Dumping file %s to the
                printer ...", fName );
    else
        sprintf( outLine, "Dumping file %s to file %s ...",
                fName, outName );
    SayHelp( outLine );

    /* Print the file. */
    printHeader();
    currentRow = 0;
    curLoc = 0L;

    if (hexMode)
    {
        do
        {
            strcpy( inLine, "................" );

            /* First clear all values to zero. */
            for (i=0; i<16; i++)
                inLine[i+17] = 0;

            /* Now read in the characters from the file. */
            for (i=0; i<16; i++)
            {
                inLine[i+17] = getc( inF );
                if (feof(inF))
                    break;
            }

            /* Now print the results. */
            fprintf( outFile, "[%06X]", curLoc );
            for (i=0; i<16; i++)
            {
                if ( (i%4) == 0)
                    fprintf( outFile, "   ");
                fprintf( outFile, "%02x", inLine[i+17]);
                if (inLine[i+17] >= ' ')
                    inLine[i] = inLine[i+17];
            }
```

```
                fprintf( outFile, "   [%s]\n", inLine );

                currentRow++;
                if (currentRow > 55)
                {
                   fprintf( outFile, "%c", 12 ); /* Formfeed */
                   printHeader();
                   currentRow = 0;
                }
                curLoc += 16;  /* Reading 16 bytes at a time */
             } while (!feof(inF) );
             fprintf( outFile, "%c", 12 ); /* Final formfeed */
          }
          else
          {
             do
             {
                fgets( inLine, 128, inF );
                inLine[ strlen(inLine)-1 ] = 0;  /* Remove CR/LF */

                fprintf( outFile, "%s\n", inLine );
                currentRow++;
                if (currentRow > 55)
                {
                   fprintf( outFile, "%c", 12 ); /* Formfeed */
                   printHeader();
                   currentRow = 0;
                }
             } while (!feof(inF) );
             fprintf( outFile, "%c", 12 ); /* Final formfeed */
          }
          SayHelp( "Press any key to continue ...");
          Inkey();
          ClearHelp();
       }
    }

    void BaseHelp()
    {
       SayHelp( "Press any key for next screen, or ESC to quit
             viewing the file");
    }

    void printHeader( void )
    {
```

```
   int i, l;
   sprintf( outLine, "---------%s Dump of File %s ---------",
      (hexMode == modeHEX) ? "-- Hex" : " ASCII", inName );
   l = strlen( outLine );
   l = (80-l)/2;   /* Center the header. Find out how many
                      spaces we need.*/
   for (i=0; i<l; i++)
      fprintf( outFile, " " );
   fprintf( outFile, "%s\n\n", outLine );
}
```

ENHANCING THE SOLUTION

One enhancement to this File-dump Utility that would be very helpful to most users is a listing of available files; this list would appear if no filename were specified on the command line. To add this capability, you will need to add an interface to MS-DOS. This DOS-interface enhancement is shown here specifically for the DiskTutor compiler, but converting it to other compilers should not be difficult.

Tip: Most compilers, including the C DiskTutor compiler, have functions that perform directory-handling work for you; these functions are typically named something like **findfirst( )** and **findnext( )**.

The DiskTutor compiler uses the following structures, **#define** statements, and prototypes for the directory-handling functions necessary to provide the listing of available files:

```
#include <dos.h>

struct find_t {
        char reserved[21];       /* Reserved for use by
                                    DOS    */
        char attrib;             /* Attribute byte for
                                    file    */
        unsigned short wr_time;  /* Time of last write
                                    to file */
        unsigned short wr_date;  /* Date of last write
                                    to file */
```

```
        unsigned long  size;      /* Length of file in
                                     bytes    */
        char name[13];            /* NULL-terminated
                                     filename   */
};

/* File attribute constants for attribute field */

#define _A_NORMAL       0x00     /* Normal file; read/write
                                    permitted */
#define _A_RDONLY       0x01     /* Read-only file */
#define _A_HIDDEN       0x02     /* Hidden file */
#define _A_SYSTEM       0x04     /* System file */
#define _A_VOLID        0x08     /* Volume-ID entry */
#define _A_SUBDIR       0x10     /* Subdirectory */
#define _A_ARCH         0x20     /* Archive file */

unsigned _dos_findfirst( const char *path,
                         unsigned attributes,
                         struct find_t *buffer);

unsigned _dos_findnext( struct find_t *buffer);
```

Let's take a look at how all this works. The **_dos_findfirst( )** function begins reading a list of filenames, and *must* be called before calling **_dos_findnext( )**. The *path* parameter is passed into the function as the full pathname of the files to be listed. For instance, to read all the files in the current directory, you would use *.* as the *path* parameter. Or, to list all the files in the C:\CDT directory, you would use C:\CDT*.*.

The *attributes* parameter passed into **_dos_findfirst( )** is set to define the types of files that will be listed. The normal value for this parameter will be _A_NORMAL, which lists all the files in the directory. To list all the subdirectories beneath the current directory, you would use _A_SUBDIR. You can also search for hidden, system, or read-only files by using _A_HIDDEN, _A_SYSTEM, or _A_RDONLY as your attribute.

Once **_dos_findfirst( )** has been called, repeated calls to **_dos_findnext( )** return the additional files in the list. Both of these calls utilize a *buffer* containing a **struct find_t** variable. Look at the definition of this structure type above, and notice that one field in the structure is *name*, the name of the file found. The other fields are set to match the file that was found. The *attrib* field contains the type of file found, so you can process subdirectories and normal files differently by checking the attribute value of each file.

Here is a short program (Program P14-3) to display all the files in the current directory, using the TUI interface, as illustrated in Figure 14-2. Note that both calls (_dos_findfirst( ) and _dos_findnext( )) return a 0 if a file was found, or a nonzero value if no more files exist with the given name.

P14-3

```
/*
   Program:    WDIR.C    (a.k.a. P14-3.C)

   Purpose:    Show a directory listing in a screen window.

   Build Sequence:
               WCL WDIR SCREEN
            or WCL P14-3 SCREEN

   Author:     L. John Ribar
               C DiskTutor

   Date: 12/31/92
*/

#include <dos.h>
#include "screen.h"

main( int argc, char *argv[] )
{
   struct find_t aFile;      /* Last file found */
   char title[80];           /* Area to build current title */
   char filePath[80];        /* Path to search */
   int row, col;             /* Current row and column in file
                                list */
   unsigned r;               /* Return value for functions */

   /*
      Allow file selection from the command line. If no
      filename is specified, do the current directory.
   */
   if (argc < 2)
      strcpy( filePath, "*.*" );
   else
      strcpy( filePath, argv[1] );

   /* Clear the screen. */
   clrscr();
```

```
/* Put up our TUI. */
Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
sprintf( title, "Directory for %s", filePath );
SayTitle( title );

/* Draw a box for the filenames. */
DrawBox( 3, 1, 21, 78, VERT, HORIZ, NORMAL, FILL, SHADOW );

/* Now start filling the box. */
row = col = 1;   /* Start in upper-left. */
r = _dos_findfirst( filePath, _A_NORMAL + _A_SUBDIR,
                    &aFile );
while (r == 0)
{
   /*
      Print the subdirectories in a color different from
      that of the regular filenames.
   */
   if (aFile.attrib == _A_SUBDIR)
      SayColor( row+3, 2+(col-1)*14, aFile.name,
         Attr( cYellow, cBlack) );
   else
      SayColor( row+3, 2+(col-1)*14, aFile.name,
         Attr( cLightCyan, cBlack) );
   row++;
   if (row > 17)
   {
      if (col > 4)
      {
         SayHelp( "Press any key for next page ...");
         Inkey();
         ClearHelp();
         Scroll( 4, 2, 20, 77, 0, NORMAL ); /* Clear
                                               window. */
         row = col = 1;   /* Reset location, */
      }
      else
      {
         row = 1;
         col++;
      }
   }

   /* Now get next file. */
   r = _dos_findnext( &aFile );
```

```
    }
    SayHelp( "Press any key to quit ... ");
    Inkey();      /* Wait for a keypress. */
    clrscr();     /* Clean up the screen. */
}
```

A version of DUMP that includes this file-listing functionality can be constructed in several ways. You might show the directory only if no files were entered on the command line. Or you might add another switch (command) on the command line, such as /D for directory, to specifically ask for a directory. Then you could ask the user for the file to be listed, and continue the rest of the program. These enhancements are left to the reader to complete.

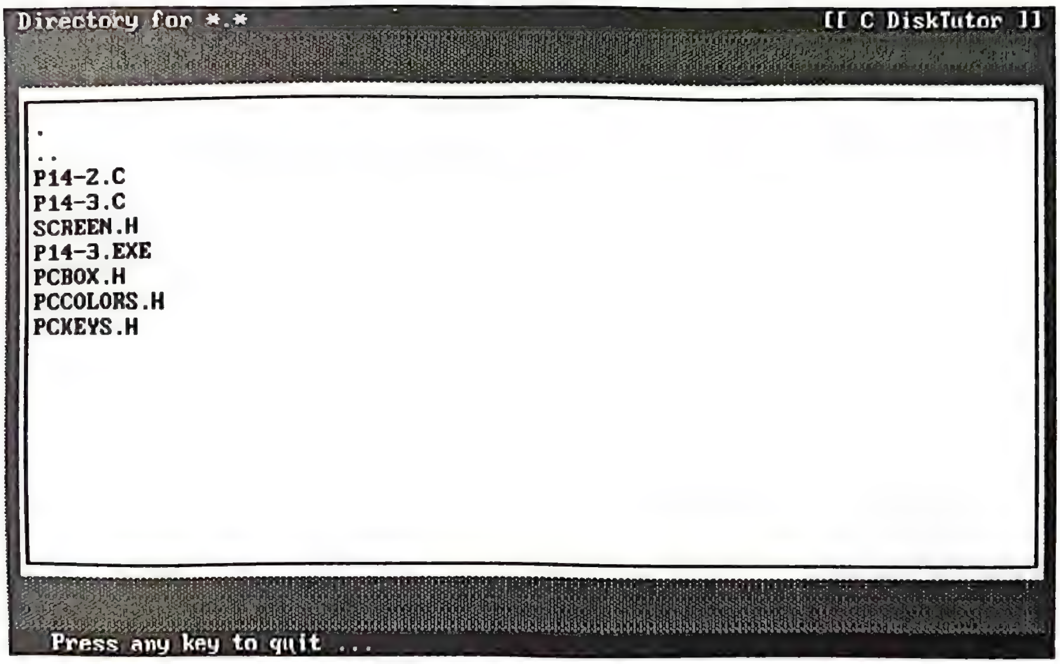


FIGURE 14-2
Directory listing displayed using the TUI Library

C H A P T E R

CASE STUDY 3: AN ELECTRONIC ADDRESS BOOK

The requirements for Case Study 3, the Electronic Address Book, were presented in Chapter 11, and are listed again here. In this chapter, you will see the program developed step by step, with additional functions added in small increments. The requirements for the Electronic Address Book are as follows:

- Use the TUI Library to create a friendly interface for the user (as illustrated in Figure 15-1).
- Allow adding new names to the data file (see Figure 15-2).
- Allow searching for a person's name in the data file, and display the person's information when it is found (see Figure 15-3). If the name is not found, display an appropriate message.
- Allow printing of mailing labels and a sorted telephone number listing (as shown in Figure 15-4 and 15-5).
- When printing labels, allow user to first print a test label, in order to adjust the printer if necessary.

- Allow deletion of names from the file. Once a name is deleted, that name should not appear in printouts or in the results of a search.
- *Optional.* Allow printing to either the printer, a disk file, or the screen.

This project makes extensive use of the Screen (TUI) Library (developed in Chapter 8) and the file-handling functions (discussed in Chapter 9), as demonstrated in Figures 15-1, 15-2, and 15-3.

STARTING THE PROJECT

Since this project allows multiple commands to be entered by the user, the program will be designed as a big loop. Within the loop, the program

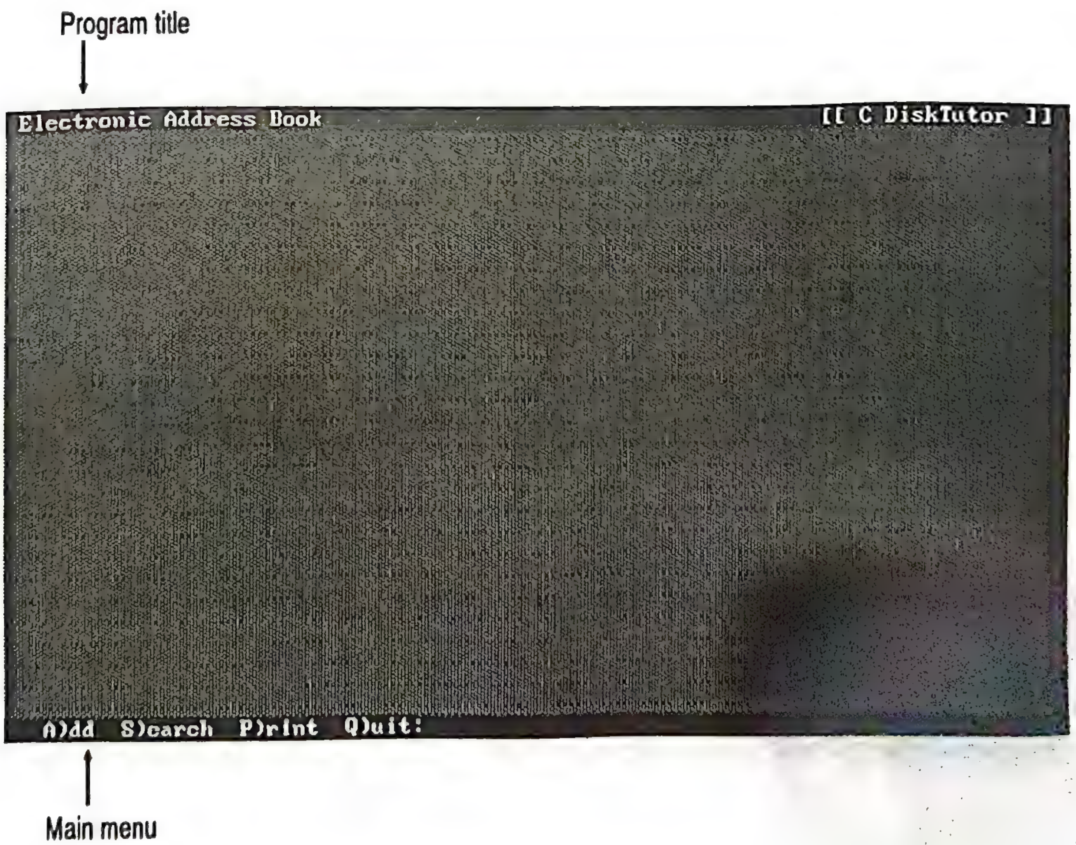


FIGURE 15-1
The opening screen of the Electronic Address Book

will read and act upon each command, one at a time. Pseudocode for the program might look like this:

```
Set up Defaults
Open the Data File
Display the Main Menu
WHILE Menu Selection Was NOT Quit DO
   SWITCH Selection OF
      CASE Add : Add a User
      CASE Search : Search For a User
      CASE Print:
         Display Print Menu
         Select Print Option
         Print Correct Option
   END SWITCH
END WHILE Loop
Close the Data File
```

Of course, the command processing for this program is much more complex than represented in the pseudocode. In fact, because of the size of the project, the code has been divided into three files:

Program File Name	A.K.A. File Name	Purpose
P15-1.C	BOOK.C	Main loop, print routines, and search routines
P15-2.H	BOOK.H	Structures and prototypes used throughout the project
P15-3.C	BOOKFUNC.C	File and screen manipulation functions

FILE-HANDLING FUNCTIONS

There are only a few major file-handling tasks that must be managed. The program must

1. Open the file.
2. Read the header record. This record (or structure) is kept at the beginning of the file (position 0); it contains the number of records currently in the file, and the number of records that are currently active (have not been deleted).

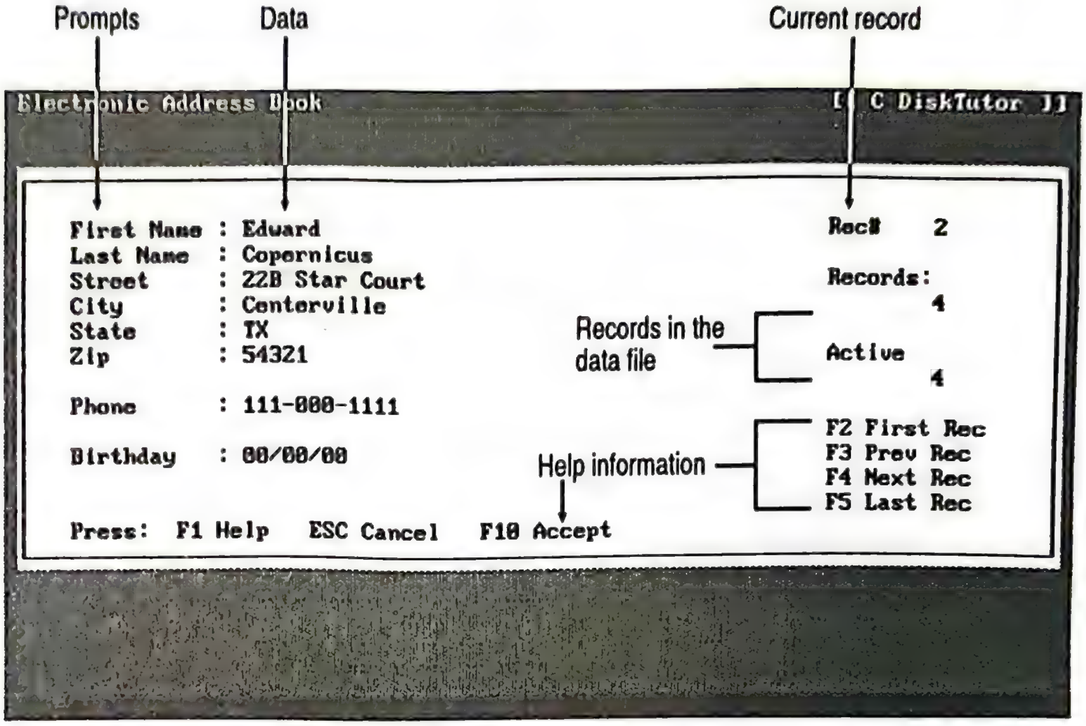


FIGURE 15-2
Adding a new record into the address book

3. Read the data records. The location for each data record in the file is calculated by using the following formula:

   Location in file = (Size of header)
   + (Record number) * (Size of data record)

4. Write data records to the file, using the location determined by the formula shown in step 3.
5. Write the header record back to the file with the updated values, if the user has made changes to the file (additions, deletions, and so on).
6. Close the data file.

To aid the design of these functions, they have been placed into the specific C functions shown in the following table.

File-handling Function	Purpose
OpenData()	Opens the data file and reads the header record. If the file does not exist, it will be created, and a header record generated.
loadRec()	Reads a specific data record from the data file.
saveRec()	Saves a specific data record back to the data file.
CloseData()	Updates the header record on disk, and then closes the data file.

Also, the following additional functions were created to facilitate easy movement through the files.

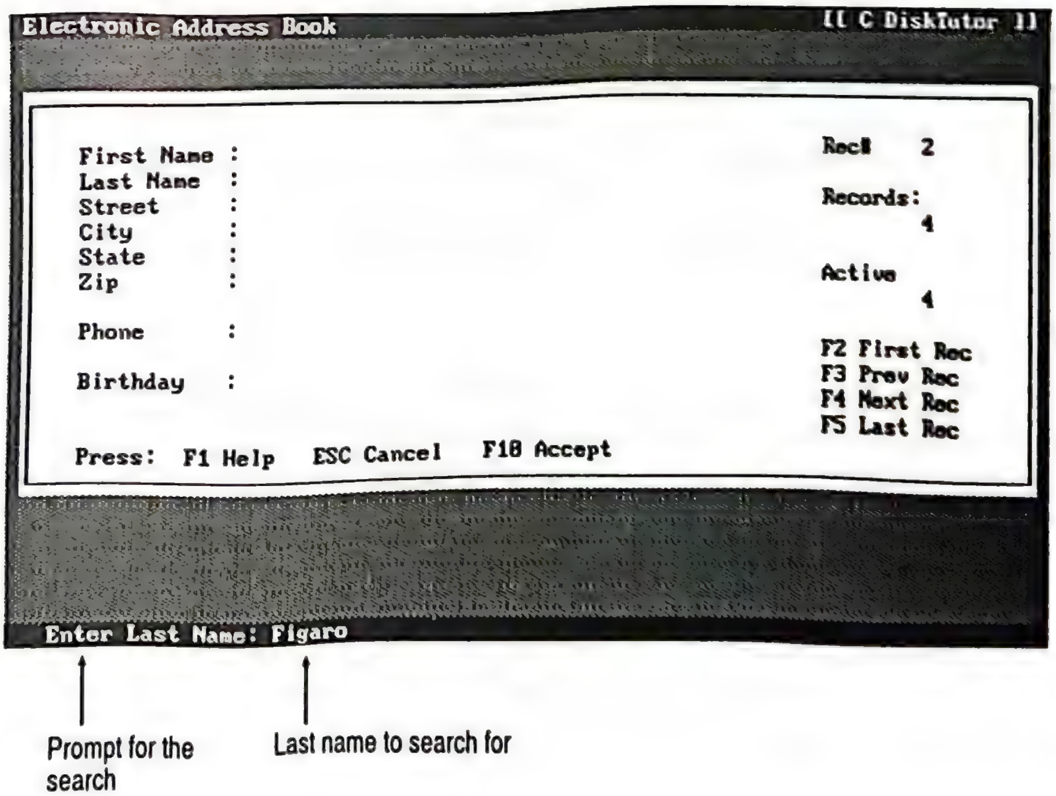


FIGURE 15-3
Searching for a name record in the address book

File-handling Function	Purpose
FirstRec( )	Reads the first data record.
NextRec( )	Reads the next data record in sequence.
PrevRec( )	Reads the previous data record in the file.
LastRec( )	Reads the final data record in the file.
FileStatus( )	Returns the number of total records in the file, along with the number currently in use (not deleted).

SCREEN-HANDLING FUNCTIONS

In the same fashion as the file-handling functions, the screen-handling operations have been organized into small functions. The following functions perform most of the screen display actions needed, once the TUI front-end is displayed:

Screen-handling Function	Purpose
ShowEditWin( )	Displays the address window on the TUI background.
ClearEditWin( )	Clears the data from an address record out of the window.
EraseEditWin( )	Removes the editing window from the screen.

```
Copernicus      Edward      111-000-1111
Einstein        Albert      101-202-EMC2
Figaro          Fred        123-456-7890
Smith           Sammy           222-2222
```

Sorted by last name

FIGURE 15-4
Alphabetical phone list generated by this case-study program

ShowPerson( )	Displays an address record in the editing window.
EditPerson( )	Allows the editing of an address record.

And to make interface with the user simpler, these two input functions were created:

Screen-handling Function	Purpose
GetString( )	Allows the entry of a data field, keeping the user within a given size, and allowing movement using the arrow keys, BACKSPACE, HOME, and END keys.
GetDate( )	For reading dates from the user, allows the same actions as for **GetString( )**.

The program also makes extensive use of the TUI functions **SayHelp( )** and **Inkey( )**.

PRINTING THE DATA FILE

Two reports are needed in this case study. In both cases, the address records are read from the data file by using **FirstRec( )** to read the first record, and then repeated calls to **NextRec( )** until the end of the file is reached.

The following functions are used in the printing process:

Printing Functions	Purpose
printLabels( )	Prints mailing labels for each active address record in the data file.
printPhoneList( )	Prints an alphabetical phone list, using all active records in the file.

The **printPhoneList( )** function is especially interesting, because it uses the **qsort( )** function (found in stdlib.h) to sort the data for the phone list printout. As seen in the following prototype, **qsort( )** requires four parameters:

```
#include <stdlib.h>

void qsort( void *base, size_t num, size_t width,
            int (*compar) ( const void *, const void * )  );
```

```
FirstName......LastName.......
Street..................
City...........,ST Zip..

Albert Einstein
10 Smart Row
Cowabunga, CA 91872

Edward Copernicus
22B Star Court
Centerville, TX 54321

Fred Figaro
98-2C Songbird Lane
Sing City, NV 84736

Sammy Smith
1 White Street
Los Angeles, CA 86954
```

Test label

Data labels

FIGURE 15-5

Mailing labels generated by this case-study program

The first three parameters are rather simple: the address of the array to be sorted, the number of items in the array, and the size of each item. The fourth parameter asks for a function that compares two items, and returns the results shown here:

Return Value of Comparison Function	Meaning
< 0	First item is less than second item
0	First item equals second item
> 0	First item is greater than second item

The array itself was created using **calloc( )** to allocate enough space, and the array was then filled using **FirstRec( )** and repeated calls to **NextRec( )**, capturing only the active records.

At the end of each report, to complete it, **fflush( )** is called to flush any buffered data to the printer.

> Tip: One possible addition to this program would be to accelerate the search for address records. This can be accomplished in a manner similar to that shown previously with **qsort( )**. You can create a sorted array, each member of which holds the record number for an address record. Then, when searching for a particular person's name, you can use the **bsearch( )** function (in stdlib.h) to quickly scan the array for the requested address record.

THE FINAL PROGRAM

Here are the files needed to build the final program. The files are named Program P15-1, P15-2, and P15-3 (also known as BOOK.C, BOOK.H, and BOOKFUNC.C on the C DiskTutor disk). To build the program, you can use either of the following build commands:

```
WCL BOOK BOOKFUNC SCREEN
```

or

```
WCL P15-1 P15-3 SCREEN
```

Here is the code for the main program file (Program P15-1 or BOOK.C):

P15-1

```
/*
 *    Case Study 3
 *
 *    Program:   BOOK.C  (a.k.a. P15-1.C)
 *
 *    Purpose:   Maintain an Electronic Address Book.
 *
 *    Build Sequence:
 *               WCL BOOK BOOKFUNC SCREEN
```

```
 *            or WCL P15-1.C P15-3.C
 *
 *    By:          L. John Ribar
 *                 C DiskTutor
 *
 *    Date:        12/30/92
 */

/*
   Preprocessor Directives
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/************************ IMPORTANT ***********************
   If you have not renamed your files in this chapter, and are
   therefore still using the numbered file names, the following
   #include file should use the file P15-2.H, rather than
   BOOK.H.
 ************************ IMPORTANT ***********************/

#include "book.h"

/*
   Static Variables
*/
static ADDRESS anAddress;     /* For passing address records */
static BOOL editWinShowing = FALSE; /* Is the edit window up? */

/*
   Local Prototypes
*/

void ClearRecord( ADDRESS *a );     /* Set all fields empty. */
void SetupTUI( void );              /* Put TUI on the screen. */
int doExit( int errorCode );        /* Prepare to exit the
                                       program. */
BOOL Match( char *one, char *two ); /* Compare two values,
                                      converting them to
                                      lowercase so that case
                                      won't matter. */
void printLabels( void );          /* Print address labels. */
```

```
void printPhoneList( void );        /* Print phone listing in
                                       sorted order.*/

/*
   Main Processing Area
*/

int main( int argc, char *argv[] )
{
   char command;              /* Command from the user */
   char aName[16];            /* Name for searching */
   int recNum;                /* Local copy of record number */
   int status;
   BOOL done;
   char ch;
   BOOL bStatus;

   /* Put up the TUI interface. */
   SetupTUI();

   SayHelp("Loading Data File");
   if (!OpenData("Address.DAT") )
   {
      SayHelp("ERROR! Could not open data file. Press a key
             to exit...");
      Inkey();
      return doExit(1);
   }
   ClearHelp();

   do
   {
      SayHelp("A)dd  S)earch  P)rint  Q)uit: ");
      command = tolower( Inkey() );
      ClearHelp();

      switch (command)
      {
         case F1 :
            SayHelp("Please select A, D, P, or Q. Press a key
                   to continue");
            Inkey();
            break;
        case 'a':   /* Add a new record. */
           if (!editWinShowing)
           {
```

```
            ShowEditWin();
            editWinShowing = TRUE;
         }
         else
            ClearEditWin();    /* Clear previous data,
                                  if any. */
         ClearRecord( &anAddress );
         do
         {
            status = EditPerson( &anAddress );
            switch (status)
            {
               case F1 :
                  SayHelp("No help is available here!
                           Press a key...");
                  Inkey();
                  ClearHelp();
                  break;
               case F2 :
                  FirstRec( &anAddress );
                  ShowPerson( anAddress );
                  break;
               case F3 :
                  PrevRec( &anAddress );
                  ShowPerson( anAddress );
                  break;
               case F4 :
                  NextRec( &anAddress );
                  ShowPerson( anAddress );
                  break;
               case F5 :
                  LastRec( &anAddress );
                  ShowPerson( anAddress );
                  break;
            }
         } while (status != FALSE);
         ClearEditWin();
         break;

      case 's':   /* Search for a record. */
         /* Ask for name to find. */
         memset( aName, 0, 16);     /* Start without
                                       a name. */
         SayHelp("Enter Last Name: ");
         GetString( 24, 20, 15, aName );
```

```
bStatus = FirstRec( &anAddress );
recNum = 0;    /* No match found yet. */
while (bStatus)
{
   if (anAddress.active)
      if ( Match( anAddress.lastName, aName ) )
      {
         recNum = anAddress.fileLoc;
         break;   /* To exit the loop */
      }
   bStatus = NextRec( &anAddress );
}

if (recNum)        /* A match was found. */
{
   /* Display the record, if found. */
   if (!editWinShowing)
   {
      ShowEditWin();
      editWinShowing = TRUE;
   }
   else
      ClearEditWin();   /* Clear previous data,
                           if any. */

   /*
      Clear off keys that are not available
      in Search mode (F2-F5).
   */
   Say( 13, 62, "             ");
   Say( 14, 62, "            ");
   Say( 15, 62, "            ");
   Say( 16, 62, "            ");

   ShowPerson( anAddress );
   done = FALSE;
   while (!done)
   {
      SayHelp("E)dit, D)elete, N)ext or Q)uit? ");
      ch = Inkey();
      switch ( tolower(ch) )
      {
         case 'e':
            status = EditPerson( &anAddress );
            done = TRUE;
```

```
                    break;
                case 'd':
                    SayHelp("Are you sure? ");
                    ch = Inkey();
                    if ( tolower(ch) == 'y' )
                    {
                        anAddress.active = FALSE;
                        saveRec( anAddress.fileLoc,
                                 &anAddress );
                    }
                    done = TRUE;
                    break;
                case 'n':
                    if (NextRec( &anAddress ))
                        ShowPerson( anAddress );
                    else
                    {
                        SayHelp("No more records. Press
                                 a key ...");
                        Inkey();
                    }
                    /*
                        Notice that done is not set to
                        TRUE, because the user may want to
                        edit or delete this new record,
                        or move forward to another record.
                    */
                    break;
                case 'q':
                    done = TRUE;
                    break;
                default:
                    break;
            }
        }
        /* Now redisplay the help. */
        Say( 13, 62, "F2 First Rec");
        Say( 14, 62, "F3 Prev Rec");
        Say( 15, 62, "F4 Next Rec");
        Say( 16, 62, "F5 Last Rec");

    }
    else
    {
        SayHelp("No match found. Press any key
```

```
                          to continue ...");
                  Inkey();
               }
               break;

            case 'p':   /* Print the records. */
               if (editWinShowing)
               {
                  EraseEditWin();
                  editWinShowing = FALSE;
               }

               /* Display print options. */
               SayHelp( "Print A)ddress labels or P)hone
                        listing? ");
               ch = Inkey();
               switch ( tolower(ch) )
               {
                  case 'a':
                     printLabels();
                     break;
                  case 'p':
                     printPhoneList();
                     break;
                  default:
                     break;
               }
               break;

            default:
               break;
         };
   } while (command != 'q');

   return doExit(0);
}

/*
   Local Functions
*/

/*
   ClearRecord() clears an ADDRESS record.
*/
void ClearRecord( ADDRESS *a )
```

```
{
   memset( (char *) a, 0, sizeof( ADDRESS ) );
}

/*
   SetupTUI() initializes the TUI screen.
*/
void SetupTUI( void )
{
   /* First, clear the screen. */
   clrscr();

   /* Put up our TUI. */
   Backdrop( DARK_FILL, Attr( cWhite, cBlue ) );
   SayTitle( "Electronic Address Book");
   ClearHelp();      /* For consistent appearance */
}

/*
   doExit() prepares to end the program, taking care of
   closing the data files, etc.
*/
int doExit( int errorCode )
{
   CloseData();   /* Try to close data file. */
   clrscr();      /* Clear the screen. */
   return errorCode;
}

/*
   Match() compares two string values, converting them
   to lowercase so that case will not make a difference.
   Comparison stops if either
      1) End of one word is reached - returns TRUE
      2) Difference is found - returns FALSE
*/
BOOL Match( char *one, char *two )
{
   int pos;       /* Position in the words */

   pos = 0;       /* Start at the beginning */
   for (;;)
   {
      if ( (one[pos] == 0) || (two[pos] == 0) )
         return TRUE;
```

```
      if ( tolower(one[pos]) != tolower(two[pos]) )
         return FALSE;
      pos++;          /* Check next position. */
   }
}

/*
   printLabels() prints out mailing labels for the names in
   the data file. This function assumes six lines per label
   (standard for 3/4" labels) on a continuous roll.
*/
void printLabels( void )
{
   char ch;
   BOOL bStatus;

   /* First, print a test label (or a few!) for alignment. */
   SayHelp("Print a test label? (Y/N) ");
   ch = Inkey();
   while ( tolower(ch) == 'y' )
   {
      fprintf(stdprn, "\nFirstName...... LastName.......\n");
      fprintf(stdprn, "Street..................\n");
      fprintf(stdprn, "City..........., ST Zip..\n\n\n");
      SayHelp("Print another test label? (Y/N) ");
      ch = Inkey();
   }

   bStatus = FirstRec( &anAddress );
   while (bStatus)
   {
      if (anAddress.active)
      {
         fprintf(stdprn, "\n%s %s\n", anAddress.firstName,
                 anAddress.lastName);
         fprintf(stdprn, "%s\n", anAddress.street);
         fprintf(stdprn, "%s, %s  %s\n\n\n", anAddress.city,
           anAddress.state,
            anAddress.zip);
      }
      bStatus = NextRec( &anAddress );
   }
   fflush( stdprn );                /* Flush all output. */
}
```

```
/*
   Declare structure and function needed for qsort(), used to
   sort the phone number printout.
*/

typedef struct sortItem
{
   char lastName[16];
   char firstName[16];
   char phone[13];
} SortItem;

static int SortAnItem( SortItem *one, SortItem *two )
{
   int status;

   /* Check last names. */
   status = strcmp( one->lastName, two->lastName );
   if (status != 0)
      return status;

   /* Last names were the same, so check first names. */
   status = strcmp( one->firstName, two->firstName );
   return status;
}

/*
   printPhoneList() prints a sorted telephone number list,
   showing each person in the database with their phone
   number. Sorting is done with the qsort() function from the
   stdlib.h header file.
*/
void printPhoneList( void )
{
   SortItem *sList;      /* Pointer to sort list */
   HEADER aHeader;       /* Data file header info */
   BOOL bStatus;
   int recNum;           /* Location in allocated array */
   int i;                /* Index counter variable */

   if (!FileStatus( &aHeader ) )
      return;      /* Data file is not open! */

   /* Allocate and clear the array of size needed. */
   sList = calloc( sizeof(SortItem), aHeader.RecordsInUse );
```

```
   /* Load the array from the data file. */
   recNum = 0;     /* Array member to use */
   bStatus = FirstRec( &anAddress );
   while (bStatus)
   {
      if (anAddress.active)
      {
         strcpy( sList[recNum].lastName, anAddress.lastName );
         strcpy( sList[recNum].firstName, anAddress.firstName );
         strcpy( sList[recNum].phone, anAddress.phone );
         recNum ++;
         if (recNum >= aHeader.RecordsInUse)
            break;      /* Too many records found. */
      }
      bStatus = NextRec( &anAddress );
   }

   /* Now sort the array. */
   qsort( sList, recNum, sizeof(SortItem), SortAnItem );

   /* Print the results. */
   for (i = 0; i < recNum; i++ )
      fprintf( stdprn, "%-15s %-15s %12s\n",
sList[i].lastName,
         sList[i].firstName, sList[i].phone );

   fprintf( stdprn, "%c", 12 );      /* Formfeed when done */
   fflush( stdprn );                 /* Flush all output */
}
```

Here is the code for the header file (Program P15-2 or BOOK.H):

P15-2

```
/*
 *    Case Study 3
 *
 *    File:       BOOK.H  (a.k.a. P15-2.H)
 *
 *    Purpose:    Header file for Book.C, an Electronic
                  Address Book
 *
 *    By:         L. John Ribar
 *                C DiskTutor
 *
 *    Date:       12/30/92
```

```
 */

#include "screen.h"

/*
   Generic Definitions
*/

typedef int BOOL;
#define TRUE 1
#define FALSE 0

/*
   A Generic Date Structure
*/

typedef struct aDate
{
   int month;
   int day;
   int year;
} DATE;

/*
   Header record for the data file. Occurs at location 0L in
   the file, and is updated each time the file is closed.
*/

typedef struct aHeader
{
   int TotalRecords;
   int RecordsInUse;
} HEADER;

/*
   Data record for each item in the address book
*/

typedef struct anAddress
{
   BOOL active;           /* Is this record in use? */
   char firstName[16];
   char lastName[16];
   char street[26];
   char city[16];
```

```
    char state[3];
    char zip[6];
    char phone[13];
    DATE birthday;
    long int fileLoc;    /* 0 = new record, else location
                            in file */
} ADDRESS;

/*
    Prototypes for Functions in BOOKFUNC.C
*/

/*
    GetString() reads a string from the user at the row and
    column specified. Returns TRUE if a string was entered,
    FALSE if Esc was pressed. Other return values are special
    keys pressed (UpArrow, DownArrow, PGUP, PGDN, etc.).
*/
int GetString( int row, int col, int maxLen, char *string );

/*
    GetDate() reads a date from the user at the row and column
    specified. Returns TRUE if a valid date was entered,
    otherwise it returns FALSE.
*/
int GetDate( int row, int col, DATE *date );

/*
    ShowEditWin() displays the display/edit window used for
    ADDRESS records.
*/
void ShowEditWin( void );

/*
    ClearEditWin() clears a data record from the display/edit
    window.
*/
void ClearEditWin( void );

/*
    EraseEditWin() erases the display/edit window from the
    screen.
*/
void EraseEditWin( void );
```

```
/*
   ShowPerson() displays a record on the screen.
*/
void ShowPerson( ADDRESS anAddress );

/*
   EditPerson() allows the editing of an ADDRESS record.
   Returns one of the following values:
      TRUE      Edit completed; changes were made to the
                record.
      FALSE     Editing session was cancelled
      PGUP, PGDN  Returned if these keys were pressed.
                Signals that the user wants to move to the
                previous or next address.
*/
int EditPerson( ADDRESS *anAddress );

/*
   FirstRec() returns the first active record in the file.
   LastRec() returns the last active record in the file.
   NextRec() and PrevRec() return the next and previous
   records in the file. All four functions return TRUE if a
   record was found, FALSE if not.
*/
BOOL FirstRec( ADDRESS *anAddress );
BOOL LastRec( ADDRESS *anAddress );
BOOL NextRec( ADDRESS *anAddress );
BOOL PrevRec( ADDRESS *anAddress );

/*
   OpenData() opens the data file. CloseData() closes the
   data file.
   Both functions return TRUE upon success, FALSE on failure.
*/
BOOL OpenData( char *fileName );
BOOL CloseData( void );

/*
   FileStatus() returns information about the data file in
   use, in the form of a copy of the header record. It
   returns TRUE if the file has been opened, FALSE if not.
*/
BOOL FileStatus( HEADER *aHeader );

/*
```

```
   loadRec() and saveRec() are used to actually read and
   write the address records in the data file.
*/
void loadRec( int recNum, ADDRESS *anAddress );
void saveRec( int recNum, ADDRESS *anAddress );
```

Finally, here is Program P15-3 (or BOOKFUNC.C):

P15-3

```
/*
 *     Case Study 3
 *
 *     File:        BOOKFUNC.C  (a.k.a. P15-3.C)
 *
 *     Purpose:     Support functions for BOOK.C, an electronic
 *                  address book
 *
 *     By:          L. John Ribar
 *                  C DiskTutor
 *
 *     Date:        12/30/92
 */

#include <stdio.h>
#include "book.h"

/*
   Local (Static) Variables
*/
static BOOL FileIsOpen = FALSE;     /* Has the data file been
                                       opened? */
static HEADER myHeader;             /* Current header
                                       information */
static FILE *dataFile = NULL;       /* Data file handle */
static int currentRec = 1;          /* Last record number
                                       read */

static void ShowStatus( void );             /* Prototype */

/*
   GetString() reads a string from the user at the row and
   column specified. Returns TRUE if a string was entered,
   FALSE if Esc was pressed. Other return values are special
   keys pressed (UpArrow, DownArrow, PGUP, PGDN, etc.).
*/
```

```
int GetString( int row, int col, int maxLen, char *string )
{
   int i;       /* Index counter */
   int pos;     /* Position in the string */
   char ch;     /* Character retrieved from user */
   int retVal;  /* Value to return */
   BOOL done;   /* Done processing yet? */

   pos = 0;        /* Start at the beginning of the string.*/
   retVal = TRUE;  /* Assume a good return value. */
   done = FALSE;   /* Not done editing yet! */

   while (!done)
   {
      Say( row, col, string );    /* Display the string as it
                                     exists. */
      Move( row, col+pos );       /* Move to correct cursor
                                     position.*/
      ch = Inkey();               /* Read a character. */
      switch (ch)
      {
         case LtArrow:            /* Move left if possible. */
            if (pos > 0) pos--;
            break;
         case RtArrow:            /* Move right one position. */
            if (pos < maxLen) pos++;
            break;
         case UpArrow:            /* These keys move to other
                                     fields. */
         case DnArrow:
         case PGUP:
         case PGDN:
         case F1:
         case F2:
         case F3:
         case F4:
         case F5:
         case F10:
            retVal = ch;
            done = TRUE;
            break;
         case Esc:                /* Cancel the editing. */
            retVal = FALSE;
            done = TRUE;
            break;
```

```
        case CR:                      /* Accept the current
                                         string. */
           retVal = TRUE;
           done = TRUE;
           break;
        case HOMEKey:                 /* Go to beginning of
                                         string. */
           pos = 0;
           break;
        case ENDKey:                  /* Go to end of string. */
           pos = strlen(string);
           if (pos >= maxLen)
              pos--;
           break;
        case BackSp:                  /* Erase previous character. */
           if (pos>0)
           {
              for (i=pos-1; i<maxLen-1; i++)
              {
                 Say( row, col+i, " ");
                 string[i] = string[i+1];
                 string[i+1] = 0;
              }
              string[i] = 0;
              pos--;
              Say( row, col, string );
           }
           break;
        default:
           if (pos < maxLen)
           {
              string[pos] = ch;
              pos++;
           }
           break;
     }
   }
   ShowStatus();
   return retVal;
}

/*
   GetDate() reads a date from the user at the row and column
   specified. Returns TRUE if a valid date was entered,
   otherwise it returns FALSE.
```

```
*/
int GetDate( int row, int col, DATE *date )
{
   char iM[3], iD[3], iY[3];      /* Input strings */
   char msg[9];                   /* Output display */
   int stat;
   int field;                     /* Which part is being
                                     entered? */
   BOOL done;

   stat = TRUE;                   /* Assume all goes
                                     correctly. */

   /* First write current values to strings we are using. */
   sprintf( iM, "%02d", date->month);
   sprintf( iD, "%02d", date->day);
   sprintf( iY, "%02d", date->year);

   done = FALSE;
   field = 1;
   while (!done)
   {
      /* Display the current values. */
      sprintf( msg, "%02d/%02d/%02d", date->month, date->day,
               date->year );
      Say( row, col, msg );

      switch (field)
      {
         case 1:  /* Month */
            stat = GetString( row, col, 2, iM );
            if (stat != FALSE)
            {
               sscanf( iM, " %d", &date->month );
               sprintf( iM, "%02d", date->month );
            }
            break;
         case 2:  /* Day */
            stat = GetString( row, col+3, 2, iD );
            if (stat != FALSE)
            {
               sscanf( iD, " %d", &date->day );
               sprintf( iD, "%02d", date->day );
            }
            break;
```

```
    case 3:  /* Year */
       stat = GetString( row, col+6, 2, iY );
       if (stat != FALSE)
       {
          sscanf( iY, " %d", &date->year );
          sprintf( iY, "%02d", date->year );
       }
       break;
    default:
       break;
}

switch (stat)
{
    case TRUE:
       if (field == 3)
          done = TRUE;
       else
          field++;
       break;
    case FALSE:
       done = TRUE;
       stat = FALSE;
       break;
    case UpArrow:              /* These keys move to other
                                  fields.*/

    case DnArrow:
    case PGUP:
    case PGDN:
    case F1:
    case F2:
    case F3:
    case F4:
    case F5:
    case F10:
       /* Return the stat received.*/
       done = TRUE;
       break;
    default:
       if (field == 3)
          field = 1;
       else
          field++;
       break;
}
```

```
    }
    return stat;
}

/*
    ShowEditWin() displays the display/edit window used for
    ADDRESS records.
*/
void ShowEditWin( void )
{
    /* Draw the display boxes. */
    DrawBox( 3, 1, 18, 78, VERT, HORIZ, Attr(cWhite, cBlue),
             FILL, SHADOW );

    /* Now display the prompts and help text. */
    Say(  5, 5, "First Name :" );
    Say(  6, 5, "Last Name  :" );
    Say(  7, 5, "Street     : ");
    Say(  8, 5, "City       : ");
    Say(  9, 5, "State      : ");
    Say( 10, 5, "Zip        : ");
    Say( 12, 5, "Phone      : ");
    Say( 14, 5, "Birthday   : ");
    Say(  5, 62, "Rec#");
    Say(  7, 62, "Records:");
    Say( 10, 62, "Active");
    Say( 17, 5, "Press:  F1 Help   ESC Cancel   F10 Accept");
    Say( 13, 62, "F2 First Rec");
    Say( 14, 62, "F3 Prev Rec");
    Say( 15, 62, "F4 Next Rec");
    Say( 16, 62, "F5 Last Rec");
    ShowStatus();
}

/*
    ClearEditWin() clears a data record from the display/edit
    window.
*/
void ClearEditWin( void )
{
    Scroll( 5, 17, 14, 60, 0, Attr( cWhite, cBlue ) );
    ShowStatus();
}

/*
```

```
   EraseEditWin() erases the display/edit window from the
   screen.
*/
void EraseEditWin( void )
{
   ReplaceBack( 3, 1, 19, 79 );
}

/*
   ShowPerson() displays a record on the screen.
*/
void ShowPerson( ADDRESS anAddress )
{
   char dateOut[9];

   ClearEditWin();
   Say( 5, 18, anAddress.firstName );
   Say( 6, 18, anAddress.lastName );
   Say( 7, 18, anAddress.street );
   Say( 8, 18, anAddress.city );
   Say( 9, 18, anAddress.state );
   Say(10, 18, anAddress.zip );
   Say(12, 18, anAddress.phone );
   sprintf( dateOut, "%02d/%02d/%02d", anAddress.birthday
            .month, anAddress.birthday.day, anAddress
            .birthday.year );
   Say(14, 18, dateOut );
   ShowStatus();
}

/*
   EditPerson() allows the editing of an ADDRESS record.
   Returns one of the following values:
      TRUE     Edit completed; changes were made to the
               record.
      FALSE    Editing session was cancelled.
      F2-F5    Returned if these keys were pressed. Signals
               that the user wants to move to the previous or
               next address.
*/
int EditPerson( ADDRESS *anAddress )
{
   int field;
   BOOL done;
   char ch;
```

```
int rVal;

rVal = TRUE;
done = FALSE;
field = 1;
while (!done)
{
   switch (field)
   {
      case 1:
         ch = GetString( 5, 18, 15, anAddress->firstName );
         break;
      case 2:
         ch = GetString( 6, 18, 15, anAddress->lastName );
         break;
      case 3:
         ch = GetString( 7, 18, 25, anAddress->street );
         break;
      case 4:
         ch = GetString( 8, 18, 15, anAddress->city );
         break;
      case 5:
         ch = GetString( 9, 18, 2, anAddress->state );
         break;
      case 6:
         ch = GetString(10, 18, 5, anAddress->zip );
         break;
      case 7:
         ch = GetString(12, 18, 13, anAddress->phone );
         break;
      case 8:
         ch = GetDate(14, 18, &anAddress->birthday );
         break;
      default:
         break;
   }
   switch (ch)
   {
      case TRUE:
      case DnArrow:
         if (field < 8)
            field++;
         else
            field = 1;
         break;
```

```
            case UpArrow:
               if (field > 0)
                  field--;
               else
                  field = 8;
               break;
            case Esc:
            case FALSE:
               done = TRUE;
               rVal = FALSE;
               break;
            case F2:
            case F3:
            case F4:
            case F5:
               done = TRUE;
               rVal = ch;
               break;
            case F10:
               anAddress->active = TRUE;
               if ( anAddress->fileLoc == 0) /* Append to file */
               {
                  currentRec = ++myHeader.TotalRecords;
                  myHeader.RecordsInUse++;
               }
               else
                  currentRec = anAddress->fileLoc;
               done = TRUE;
               saveRec( currentRec, anAddress );
               break;
            default:
               break;
         }
      }
   return rVal;
}

/*
   FirstRec() returns the first active record in the file.
   LastRec() returns the last active record in the file.
   NextRec() and PrevRec() return the next and previous
   records in the file. All four functions return TRUE if a
   record was found, FALSE if not.
*/
BOOL FirstRec( ADDRESS *anAddress )
```

```
{
    int r;

    if (FileIsOpen)
    {
        r = 1;
        loadRec( r, anAddress );
        while ( anAddress->active == FALSE )
        {
            if (r == myHeader.TotalRecords ) /* No active
                                                records found */

                return FALSE;
            r++;
            loadRec( r, anAddress );
        }
        anAddress->fileLoc = r; /* Save record number.*/
        currentRec = r;
        return TRUE;
    }
    else
        return FALSE;
}

BOOL LastRec( ADDRESS *anAddress )
{
    int r;

    if (FileIsOpen)
    {
        r = myHeader.TotalRecords;
        loadRec( r, anAddress );
        while ( anAddress->active == FALSE )
        {
            if (r == 1) /* No active records found.*/
                return FALSE;
            r--;
            loadRec( r, anAddress );
        }
        anAddress->fileLoc = r; /* Save record number */
        currentRec = r;
        return TRUE;
    }
    else
        return FALSE;
}
```

```
BOOL NextRec( ADDRESS *anAddress )
{
   int r;

   if (FileIsOpen)
   {
      r = currentRec +1;
      if (r > myHeader.TotalRecords)
         return FALSE;
      loadRec( r, anAddress );
      while ( anAddress->active == FALSE )
      {
         if (r >= myHeader.TotalRecords ) /* No active
                                             records found.*/
            return FALSE;
         r++;
         loadRec( r, anAddress );
      }
      if (r > myHeader.TotalRecords)
         return FALSE;
      anAddress->fileLoc = r; /* Save record number.*/
      currentRec = r;
      return TRUE;
   }
   else
      return FALSE;
}

BOOL PrevRec( ADDRESS *anAddress )
{
   int r;

   if (FileIsOpen)
   {
      r = currentRec - 1;
      if (r < 1)
         return FALSE;
      loadRec( r, anAddress );
      while ( anAddress->active == FALSE )
      {
         if (r == 1) /* No active records found.*/
            return FALSE;
         r--;
         loadRec( r, anAddress );
```

```
        }
        if (r < 1)
            return FALSE;
        anAddress->fileLoc = r; /* Save record number.*/
        currentRec = r;
        return TRUE;
    }
    else
        return FALSE;
}

/*
    OpenData() opens the data file. CloseData() closes the
    data file.
    Both functions return TRUE upon success, FALSE on failure.
*/
BOOL OpenData( char *fileName )
{
    char ch;

    dataFile = fopen( fileName, "r+" );
    if (dataFile == NULL)
    {
        SayHelp("File does not exist. Should I create a new
                one? (Y/N)");
        ch = Inkey();
        if (tolower(ch) == 'y')
        {
            myHeader.TotalRecords = myHeader.RecordsInUse = 0;
            dataFile = fopen( fileName, "w+" );
            FileIsOpen = TRUE;
            return TRUE;
        }
        else
            return FALSE;
    }

    fseek( dataFile, 0L, SEEK_SET );
    fread( &myHeader, sizeof(HEADER), 1, dataFile );
    FileIsOpen = TRUE;
    return TRUE;
}

BOOL CloseData( void )
{
```

```
    if (FileIsOpen)
    {
        fseek( dataFile, 0L, SEEK_SET );
        fwrite( &myHeader, sizeof(HEADER), 1, dataFile );
        fclose( dataFile );
        FileIsOpen = FALSE;
        return TRUE;
    }
    else
        return FALSE;
}

/*
    FileStatus() returns information about the data file in
    use, in the form of a copy of the header record. It
    returns TRUE if the file has been opened, FALSE if not.
*/
BOOL FileStatus( HEADER *aHeader )
{
    memcpy( aHeader, &myHeader, sizeof( HEADER ) );
    return FileIsOpen;
}

void loadRec( int recNum, ADDRESS *anAddress )
{
    long seekLoc;

    if (recNum == 0)
        return;
    seekLoc = sizeof(HEADER) + (recNum-1)*sizeof(ADDRESS);
    fseek( dataFile, seekLoc, SEEK_SET );
    fread( anAddress, sizeof(ADDRESS), 1, dataFile );
    anAddress->fileLoc = recNum;
    currentRec = recNum;
}

void saveRec( int recNum, ADDRESS *anAddress )
{
    long seekLoc;

    if (recNum == 0)
    {
        return;
    }
    seekLoc = sizeof(HEADER) + (recNum-1)*sizeof(ADDRESS);
```

```
    fseek( dataFile, seekLoc, SEEK_SET );
    fwrite( anAddress, sizeof(ADDRESS), 1, dataFile );
    currentRec = recNum;
    /*
       If the record is active, no change is needed. If
       inactive, reduce the number of active records by 1.
    */
    myHeader.RecordsInUse += ( anAddress->active ? 0 : -1 );
}

static void ShowStatus( void )
{
    char temp[10];
    sprintf( temp, "%4d ", currentRec );
    Say( 5, 67, temp );
    sprintf( temp, "%4d ", myHeader.TotalRecords );
    Say( 8, 67, temp );
    sprintf( temp, "%4d ", myHeader.RecordsInUse );
    Say(11, 67, temp );
}
```

SUMMARY

This is the final case study presented in this book. Congratulations—you have now completed an introductory course in C programming. Your next assignment is to continue evolving the programs and utilities developed here, as well as your ideas for other programs that you need. The best way to learn C, now that you know the basics, is to continue working with it. Create some case studies for yourself, and write programs that will help you in your daily life—both personal and professional.

And most of all, have some fun in your search for the mastery of C!

PART FOUR

APPENDIXES

A P P E N D I X A

C *DISKTUTOR* TABLES AND HEADER FILES

This appendix displays several informational tables, as well as the headers files supplied as part of the DiskTutor compiler (produced by WATCOM).

C RESERVED WORDS

The following list of words are *reserved words* for the C language. They all have special meaning within the C language, and the compiler will therefore not allow their use for naming variables or functions.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

The DiskTutor compiler, and many others, also reserve these words:

based	cdecl	export	far
fortran	huge	interrupt	loadds
near	Packed	pascal	saveregs
segment	segname	self	syscall

C SPECIAL CHARACTERS

The following nonletter and nondigit characters are part of C.

Characters	Usage
{ }	Delimits blocks of code
( )	Delimits the parameter list in a function definition or a function call
[ ]	Delimits the index of an array variable
< >	Delimits the name of standard header files
>	Greater-than comparison in equations
<	Less-than comparison in equations
==	Is-equal-to comparison in equations. This is a particularly difficult character for a beginning C programmer to master
>=	Greater-than or equal-to in equations
<=	Less-than or equal-to in equations
!=	Is-not-equal-to
!	NOT. This negates the value of the item to its right

Characters	Usage
~	Complement of a number
/*	Begins a comment
*/	Ends a comment
' '	Delimits a single character
" "	limits a string of characters. Also delimits the name of a local header file (instead of '<' and '>')
*	Used in two ways:
	1) As a multiplication symbol in equations
	2) To denote a pointer variable, or the contents of a pointer variable
+	Addition symbol in equations
-	Subtraction symbol in equations
/	Division symbol in equations
%	Modulus symbol in equations. This is similar to division, but returns the remainder after dividing the two items
++	Increments a number by one
—	Decrements a number by one
|	ORs two numbers together
||	ORs the boolean value of two expressions together
&	Used in two ways:
	1) To denote the address of the variable to its right
	2) To AND two numbers together
&&	ANDs the boolean value of two expressions together
>>	Arithmetic shift-right function
<<	Arithmetic shift-left function
	Finds EXCLUSIVE-OR of two values
?	Used in special C statements to determine which of two values to choose based on a boolean operation
:	Used in conjunction with the ? operation
#	Introduces a preprocessor directive

Characters	Usage
\	Introduces a special character in a character constant or as part of a character string
;	Ends every C statement
=	Assigns values to a variable
+=	Adds the value on the right to the variable on the left. Combination of addition and assignment functions
-=	Subtracts the value on the right from the variable on the left. Combination of subtraction and assignment functions
*=	Multiplies the value on the right to the variable on the left. Combination of multiplication and assignment functions
/=	Divides the variable on the left by the value on the right. Combination of division and assignment functions
^=	EXCLUSIVE-ORs the value on the right and the variable on the left. Combination of EXCLUSIVE-OR and assignment functions
&=	ANDs the value on the right and the variable on the left. Combination of AND and assignment functions

DISKTUTOR HEADER FILES

For complete documentation on the functions shown in these header files, refer to the manuals supplied with the full version of WATCOM C, or another professional compiler.

Note: Each header file is marked as to whether it is an ANSI header file (specified in the ANSI C standard), a POSIX header file (a recent standard that attempts to make function calls "non-operating-system-specific"), or an extension supplied by WATCOM with their compiler.

ANSI HEADER FILE ASSERT.H

```
/*
 *  assert.h
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#undef  assert

#ifdef NDEBUG
#define assert(__ignore) ((void)0)
#else
extern void __assert( int, char *, char *, int );
#define assert(expr) __assert(expr,#expr,__FILE__,__LINE__)
#endif
```

EXTENSION HEADER FILE BIOS.H

```
/*
 *  bios.h     BIOS functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1990-1991. All
    rights reserved.
 */
#ifndef _BIOS_H_INCLUDED
#pragma pack(1);

struct diskinfo_t       /* disk parameters */
{
    unsigned drive;     /* drive number    */
    unsigned head;      /* head number     */
    unsigned track;     /* track number    */
    unsigned sector;    /* sector number   */
    unsigned nsectors;  /* number of sectors to read/write/
                           compare  */
    void __far *buffer; /* buffer to read to, write from,
                           or compare  */
};

/* constants for BIOS disk access functions */
#define _DISK_RESET    0
#define _DISK_STATUS   1
#define _DISK_READ     2
```

```
#define _DISK_WRITE    3
#define _DISK_VERIFY   4
#define _DISK_FORMAT   5

/* constants for BIOS serial communications (RS-232) support */

/* serial port services */

#define _COM_INIT     0   /* init serial port */
#define _COM_SEND     1   /* send character */
#define _COM_RECEIVE  2   /* receive character */
#define _COM_STATUS   3   /* get serial port status */

/* serial port initializers. One and only one constant from
 * each of the following four groups - character size, stop
 * bit, parity, and baud rate - must be specified in the
 * initialization byte.
 */

/* character size initializers */

#define _COM_CHR7   2   /* 7 bits characters */
#define _COM_CHR8   3   /* 8 bits characters */

/* stop bit values - on or off */

#define _COM_STOP1   0   /* 1 stop bit */
#define _COM_STOP2   4   /* 2 stop bits */

/*  parity initializers */

#define _COM_NOPARITY     0/* no parity */
#define _COM_ODDPARITY    8/* odd parity */
#define _COM_SPACEPARITY 16/* space parity */
#define _COM_EVENPARITY  24/* even parity */

/*  baud rate initializers */

#define _COM_110      0  /* 110 baud */
#define _COM_150     32  /* 150 baud */
#define _COM_300     64  /* 300 baud */
#define _COM_600     96  /* 600 baud */
#define _COM_1200   128  /* 1200 baud */
#define _COM_2400   160  /* 2400 baud */
#define _COM_4800   192  /* 4800 baud */
#define _COM_9600   224  /* 9600 baud */
```

```
/* keyboard services */

#define _KEYBRD_READ         0   /* read next character from
                                    keyboard */
#define _KEYBRD_READY        1   /* check for keystroke */
#define _KEYBRD_SHIFTSTATUS 2   /* get current shift
                                    key status */

/* services for enhanced keyboards */

#define _NKEYBRD_READ        0x10 /* read next character from
                                     keyboard */
#define _NKEYBRD_READY       0x11 /* check for keystroke */
#define _NKEYBRD_SHIFTSTATUS 0x12 /* get current shift key
                                     status */

/* printer services */

#define _PRINTER_WRITE  0    /* write character to printer */
#define _PRINTER_INIT   1    /* initialize printer */
#define _PRINTER_STATUS 2    /* get printer status */

/* time-of-day services */

#define _TIME_GETCLOCK  0   /* get current clock count */
#define _TIME_SETCLOCK  1   /* set current clock count */


unsigned short _bios_disk(unsigned __cmd,struct diskinfo_t
                          *__diskinfo);
unsigned short _bios_equiplist(void);
unsigned short _bios_keybrd(unsigned __cmd);
unsigned short _bios_memsize(void);
unsigned short _bios_printer(unsigned __cmd,unsigned
                             __port,unsigned __data);
unsigned short _bios_serialcom(unsigned __cmd,unsigned
                               __port,unsigned __data);
int            _bios_timeofday(int __cmd, long *__timeval );


#ifdef __INLINE_FUNCTIONS__
#pragma aux _bios_equiplist = 0xcd 0x11 value [ax];
#pragma aux _bios_keybrd    = 0x88 0xe6 0xcd 0x16 0x75 0x09 \
                              0x80 0xe6 0x0f \
                              0xfe 0xce 0x75 0x02 0x29 0xc0 \
                              parm [ah] value [ax] modify [dx];
```

```
#pragma aux  _bios_memsize  = 0xcd 0x12 value [ax];
#pragma aux  _bios_printer  = 0xcd 0x17 0x8a 0xc4 0xb4 0x00 \
                              parm [ah] [dx] [al] value [ax];
#pragma aux _bios_serialcom = 0xcd 0x14 parm [ah] [dx] [al]
                              value [ax];
#endif

#pragma pack();
#define _BIOS_H_INCLUDED
#endif
```

EXTENSION HEADER FILE CONIO.H

```
/*
 *  conio.h, Console and Port I/O functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _CONIO_H_INCLUDED

#ifdef __INLINE_FUNCTIONS__
 extern unsigned inp(unsigned __port);
 extern unsigned inpw(unsigned __port);
 extern unsigned outp(unsigned __port, unsigned __value);
 extern unsigned outpw(unsigned __port,unsigned __value);
 #define inp(__x)        _inline_inp(__x)
 #define inpw(__x)       _inline_inpw(__x)
 #define outp(__x,__y)   _inline_outp(__x,__y)
 #define outpw(__x,__y) _inline_outpw(__x,__y)
#endif

extern char *cgets(char *__buf);
extern int cputs(const char *__buf);
extern int cprintf(const char *__fmt,...);
extern int cscanf(const char *__fmt,...);
extern int getch(void);
extern int getche(void);
extern unsigned inp(unsigned __port);
extern unsigned inpw(unsigned __port);
extern int kbhit(void);
extern unsigned outp(unsigned __port, unsigned __value);
extern unsigned outpw(unsigned __port,unsigned __value);
extern int putch(int __c);
extern int ungetch(int __c);
```

```
#define _CONIO_H_INCLUDED
#endif
```

ANSI HEADER FILE CTYPE.H

```
/*
 *  ctype.h      Character Handling
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _CTYPE_H_INCLUDED

#define _LOWER  0x80
#define _UPPER  0x40
#define _DIGIT  0x20
#define _XDIGT  0x10
#define _PRINT  0x08
#define _PUNCT  0x04
#define _SPACE  0x02
#define _CNTRL  0x01

extern int isalnum(int);
extern int isalpha(int);
extern int iscntrl(int);
extern int isdigit(int);
extern int isgraph(int);
extern int islower(int);
extern int isprint(int);
extern int ispunct(int);
extern int isspace(int);
extern int isupper(int);
extern int isxdigit(int);
extern int tolower(int);
extern int toupper(int);


#ifndef NO_EXT_KEYS   /* extensions enabled */
 extern  int isascii(int);
 #define isascii(__c) ((unsigned)(__c) <= 0x7f)
#endif

#ifdef M_I86HM
extern const char __far _IsTable[257];
#else
```

```
extern const char __near _IsTable[257];
#endif

#define isalnum(__c) (_IsTable[(__c)+1] & (_LOWER|_UPPER|
                                          _DIGIT))
#define isalpha(__c) (_IsTable[(__c)+1] & (_LOWER|_UPPER))
#define iscntrl(__c) (_IsTable[(__c)+1] & _CNTRL)
#define isdigit(__c) (_IsTable[(__c)+1] & _DIGIT)
#define isgraph(__c) ((_IsTable[(__c)+1] & (_PRINT|_SPACE))
                                           == _PRINT)
#define islower(__c) (_IsTable[(__c)+1] & _LOWER)
#define isprint(__c) (_IsTable[(__c)+1] & _PRINT)
#define ispunct(__c) (_IsTable[(__c)+1] & _PUNCT)
#define isspace(__c) (_IsTable[(__c)+1] & _SPACE)
#define isupper(__c) (_IsTable[(__c)+1] & _UPPER)
#define isxdigit(__c) (_IsTable[(__c)+1] & _XDIGT)

#define _CTYPE_H_INCLUDED
#endif
```

EXTENSION HEADER FILE DIRECT.H

```
/*
 *  direct.h, Defines the types and structures used by the
    directory routines
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _DIRECT_H_INCLUDED
#pragma pack(1);

#ifndef _TYPES_H_INCLUDED
 #include <sys/types.h>
#endif

#define NAME_MAX  12      /* 8 chars + '.' +  3 chars */

typedef struct dirent {
    char d_dta[ 21 ];              /* disk transfer area */
    char d_attr;                   /* file's attribute */
    unsigned short int d_time;     /* file's time */
    unsigned short int d_date;     /* file's date */
    long d_size;                   /* file's size */
    char d_name[ NAME_MAX + 1 ];   /* file's name */
```

```
    ino_t d_ino;                    /* serial number
                                       (not used) */
    char d_first;                   /* flag for 1st time */
} DIR;

/* File attribute constants for d_attr field */

#define _A_NORMAL 0x00  /* Normal file - read/write
                           permitted */
#define _A_RDONLY 0x01  /* Read-only file */
#define _A_HIDDEN 0x02  /* Hidden file */
#define _A_SYSTEM 0x04  /* System file */
#define _A_VOLID  0x08  /* Volume-ID entry */
#define _A_SUBDIR 0x10  /* Subdirectory */
#define _A_ARCH   0x20  /* Archive file */


int    chdir( const char *__path );
int    closedir( DIR * );
char   *getcwd( char *__buf, unsigned __size );
int    mkdir( const char *__path );
DIR    *opendir( const char * );
struct dirent *readdir( DIR * );
int    rmdir( const char *__path );

#pragma pack();
#define _DIRECT_H_INCLUDED
#endif
```

EXTENSION HEADER FILE DOS.H

```
/*
 *  dos.h       Defines the structs and unions used to handle
 *              the input and output registers for the DOS
 *              and 386 DOS Extender interface routines.
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _DOS_H_INCLUDED

#include <i86.h>

#if defined( __WINDOWS_386__ )
#define __far
```

```
#endif
#pragma pack(1);

/* dosexterror struct */

struct DOSERROR {
        int exterror;
        char class;
        char action;
        char locus;
};

struct dosdate_t {
        unsigned char day;      /* 1-31 */
        unsigned char month;    /* 1-12 */
        unsigned short year;    /* 1980-2099 */
        unsigned char dayofweek;/* 0-6 (0=Sunday) */
};

struct dostime_t {
        unsigned char hour;     /* 0-23 */
        unsigned char minute;   /* 0-59 */
        unsigned char second;   /* 0-59 */
        unsigned char hsecond;  /* 1/100 second; 0-99 */
};

struct find_t {
        char reserved[21];      /* reserved for use by DOS    */
        char attrib;            /* attribute byte for file    */
        unsigned short wr_time; /* time of last write to file */
        unsigned short wr_date; /* date of last write to file */
        unsigned long  size;    /* length of file in bytes    */
        char name[13];          /* NULL-terminated filename   */
};

/* Critical error handler equates for _hardresume result
   parameter */

#define _HARDERR_IGNORE 0       /* Ignore the error */
#define _HARDERR_RETRY  1       /* Retry the operation */
#define _HARDERR_ABORT  2       /* Abort the program */
#define _HARDERR_FAIL   3       /* Fail the system call in
                                   progress */

/* File attribute constants for attribute field */
```

```
#define _A_NORMAL       0x00    /* Normal file - read/write
                                   permitted */
#define _A_RDONLY       0x01    /* Read-only file */
#define _A_HIDDEN       0x02    /* Hidden file */
#define _A_SYSTEM       0x04    /* System file */
#define _A_VOLID        0x08    /* Volume-ID entry */
#define _A_SUBDIR       0x10    /* Subdirectory */
#define _A_ARCH         0x20    /* Archive file */

struct diskfree_t {
        unsigned short total_clusters;
        unsigned short avail_clusters;
        unsigned short sectors_per_cluster;
        unsigned short bytes_per_sector;
};


int      bdos(int __dosfn, unsigned int __dx, unsigned
              int __al);
void     _chain_intr(void (__interrupt __far *__handler)());
unsigned _dos_allocmem( unsigned __size, unsigned short
                        *__segment );
unsigned _dos_close( int __handle );
unsigned _dos_creat( const char *__path, unsigned __attr, int
                     *__handle );
unsigned _dos_creatnew( const char *__path, unsigned __attr,
                        int *__handle );
unsigned _dos_findfirst(const char *__path, unsigned
                        __attr,struct find_t *__buf);
unsigned _dos_findnext( struct find_t *__buf);
unsigned _dos_freemem( unsigned short __segment );
void     _dos_getdate( struct dosdate_t *__date );
unsigned _dos_getdiskfree( unsigned __drive, struct
                           diskfree_t *__diskspace);
void     _dos_getdrive( unsigned *__drive );
unsigned _dos_getfileattr( const char *__path, unsigned
                           *__attr );
unsigned _dos_getftime( int __handle, unsigned short *__date,
                        unsigned short *__time );
void     _dos_gettime( struct dostime_t *__time );
void     (__interrupt __far *_dos_getvect(int __intnum))();
void     _dos_keep(unsigned __retcode,unsigned __memsize);

unsigned _dos_open( const char *__path, unsigned __mode, int
                    *__handle );
unsigned _dos_read( int __handle, void __far *__buf, unsigned
```

```
                        __count, unsigned *__bytes );
unsigned _dos_setblock( unsigned __size, unsigned short
                        __segment, unsigned  *__maxsize );
unsigned _dos_setdate( struct dosdate_t *__date );
void     _dos_setdrive( unsigned __drivenum, unsigned
                        *__drives );
unsigned _dos_setfileattr( const char *__path, unsigned
                        __attr );
unsigned _dos_setftime(int __handle,unsigned short __date,
                        unsigned short __time);
unsigned _dos_settime( struct dostime_t *__time );
void     _dos_setvect(int __intnum,void (__interrupt __far
                        *__handler)());
unsigned _dos_write( int __handle, void __far *__buf,
                        unsigned __count, unsigned *__bytes );
int      dosexterr(struct DOSERROR *);
void     _harderr(int (__far *__func)(unsigned __deverr,
                        unsigned __errcode,unsigned __far
                        *__devhdr));
void     _hardresume(int __result);
void     _hardretn(int __error);
int      intdos(union REGS *, union REGS *);
int      intdosx(union REGS *, union REGS *, struct SREGS *);
void     sleep(unsigned __seconds);

#pragma pack();
#define _DOS_H_INCLUDED
#if defined( __WINDOWS_386__ )
#undef __far
#endif
#endif
```

POSIX HEADER FILE ENV.H

```
/*
 *  env.h, Environment string operations
 *
 *  Copyright (C) by WATCOM Systems Inc. 1990., All rights
    reserved.
 */
#ifndef _ENV_H_INCLUDED

/*
 *  POSIX 1003.1 Prototypes.
 */
```

```
int   clearenv( void );
char  *getenv( const char *__name );
int   setenv( const char *__name, const char *__newvalue,
int __overwrite );
#if !defined(NO_EXT_KEYS)
int   putenv( const char *__env_string );
#endif

#define _ENV_H_INCLUDED
#endif
```

ANSI HEADER FILE ERRNO.H

```
/*
 *  errno.h, Error numbers
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _ERRNO_H_INCLUDED

#define errno (*__get_errno_ptr())
extern volatile int errno;

/*
 * Error codes
 */
#define ENOENT    1   /* No such file or directory */
#define E2BIG     2   /* Arg list too big */
#define ENOEXEC   3   /* Exec format error */
#define EBADF     4   /* Bad file number */
#define ENOMEM    5   /* Not enough memory */
#define EACCES    6   /* Permission denied */
#define EEXIST    7   /* File exists */
#define EXDEV     8   /* Cross-device link */
#define EINVAL    9   /* Invalid argument */
#define ENFILE   10   /* File table overflow */
#define EMFILE   11   /* Too many open files */
#define ENOSPC   12   /* No space left on device */

/* math errors */
#define EDOM     13   /* Argument too large */
#define ERANGE   14   /* Result too large */
```

```
/* file locking error */
#define EDEADLK   15   /* Resource deadlock would occur */
#define EDEADLOCK 15   /* ... */
#define EINTR     16   /* interrupt */
#define ECHILD    17   /* Child does not exist */

#define _ERRNO_H_INCLUDED
#endif
```

POSIX HEADER FILE FCNTL.H

```
/*
 *  fcntl.h      File control options used by open
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _FCNTL_H_INCLUDED

#define O_RDONLY   0x0000   /* open for read only */
#define O_WRONLY   0x0001   /* open for write only */
#define O_RDWR     0x0002   /* open for read and write */
#define O_APPEND   0x0010   /* writes done at end of file */
#define O_CREAT    0x0020   /* create new file */
#define O_TRUNC    0x0040   /* truncate existing file */
#define O_NOINHERIT  0x0080 /* file is not inherited by child
                               process */
#define O_TEXT     0x0100   /* text file */
#define O_BINARY   0x0200   /* binary file */
#define O_EXCL     0x0400   /* exclusive open */

int  open(const char *__path,int __oflag,...);
int  sopen(const char *__path,int __oflag,int __shflag,...);

#define _FCNTL_H_INCLUDED
#endif
```

ANSI HEADER FILE FLOAT.H

```
/*
 *  float.h      Floating point functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
```

```
 */
#ifndef _FLOAT_H_INCLUDED

#define FLT_RADIX   2
#define FLT_ROUNDS  1   /* round to nearest */

/* number of base-FLT_RADIX digits in the floating point
   mantissa */
#define FLT_MANT_DIG   23
#define DBL_MANT_DIG   53
#define LDBL_MANT_DIG  53

/* number of decimal digits of precision */
#define FLT_DIG    6
#define DBL_DIG   15
#define LDBL_DIG  15

/* minimum negative integer such that FLT_RADIX raised to
   that power minus 1 is a normalized floating point number */
#define FLT_MIN_EXP    (-127)
#define DBL_MIN_EXP    (-1023)
#define LDBL_MIN_EXP   (-1023)

/* minimum negative integer such that 10 raised to that power
   is in the range of normalized floating point numbers */
#define FLT_MIN_10_EXP  (-38)
#define DBL_MIN_10_EXP  (-307)
#define LDBL_MIN_10_EXP (-307)

/* maximum integer such that FLT_RADIX raised to that power
   minus 1 is a representable floating point number */
#define FLT_MAX_EXP    127
#define DBL_MAX_EXP   1023
#define LDBL_MAX_EXP  1023

/* maximum integer such that 10 raised to that power is in
   the range of representable floating point numbers */
#define FLT_MAX_10_EXP   38
#define DBL_MAX_10_EXP  308
#define LDBL_MAX_10_EXP 308

/* maximum representable floating point number */
#define FLT_MAX    3.402823466e+38f
#define DBL_MAX    1.7976931348623156e+308
#define LDBL_MAX   DBL_MAX
```

```
/* minimum positive floating point number x such that 1.0 + x
   != 1.0 */
#define FLT_EPSILON     1.192092896e-7f
#define DBL_EPSILON     2.2204460492503131e-16
#define LDBL_EPSILON    DBL_EPSILON

/* minimum representable positive floating point number */
#define FLT_MIN     1.175494351e-38f
#define DBL_MIN     2.2250738585072016e-308
#define LDBL_MIN    DBL_MIN


#ifndef NO_EXT_KEYS       /* extensions enabled */
/*
 *  8087/80287/80387 math co-processor control information
 */

/* 80(x)87 Control Word Mask and bit definitions. */

#define MCW_EM          0x003f  /* interrupt Exception Masks */
#define EM_INVALID      0x0001  /*   invalid */
#define EM_DENORMAL     0x0002  /*   denormal */
#define EM_ZERODIVIDE   0x0004  /*   zero divide */
#define EM_OVERFLOW     0x0008  /*   overflow */
#define EM_UNDERFLOW    0x0010  /*   underflow */
#define EM_PRECISION    0x0020  /*   inexact result */

#define MCW_IC          0x1000  /* Infinity Control */
#define IC_AFFINE       0x1000  /*   affine */
#define IC_PROJECTIVE   0x0000  /*   projective */

#define MCW_RC          0x0c00  /* Rounding Control */
#define RC_NEAR         0x0000  /*   near */
#define RC_DOWN         0x0400  /*   down */
#define RC_UP           0x0800  /*   up */
#define RC_CHOP         0x0c00  /*   chop */

#define MCW_PC          0x0300  /* Precision Control */
#define PC_24           0x0000  /*    24 bits */
#define PC_53           0x0200  /*    53 bits */
#define PC_64           0x0300  /*    64 bits */

/* 80(x)87 Status Word bit definitions */

#define SW_INVALID      0x0001  /*   invalid */
#define SW_DENORMAL     0x0002  /*   denormal */
```

```
#define SW_ZERODIVIDE  0x0004  /*   zero divide */
#define SW_OVERFLOW    0x0008  /*   overflow */
#define SW_UNDERFLOW   0x0010  /*   underflow */
#define SW_INEXACT     0x0020  /*   inexact (precision) */

/* Floating-point error codes */

#define FPE_INVALID    0x81
#define FPE_DENORMAL   0x82
#define FPE_ZERODIVIDE 0x83
#define FPE_OVERFLOW   0x84
#define FPE_UNDERFLOW  0x85
#define FPE_INEXACT    0x86
#define FPE_UNEMULATED 0x87
#define FPE_SQRTNEG    0x88
#define FPE_STACKOVERFLOW 0x8a
#define FPE_STACKUNDERFLOW 0x8b
#define FPE_EXPLICITGEN 0x8c

unsigned _clear87(void);
unsigned _control87(unsigned,unsigned);
void _fpreset(void);
unsigned _status87(void);
#endif

#define _FLOAT_H_INCLUDED
#endif
```

EXTENSION HEADER FILE IO.H

```
/*
 *  io.h        Low level I/O routines that work with file
 *              handles
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
 *  rights reserved.
 */
#ifndef _IO_H_INCLUDED

/* Symbolic constants for the access() function */

#define R_OK    4       /*  Test for read permission    */
#define W_OK    2       /*  Test for write permission   */
#define X_OK    1       /*  Test for execute permission */
#define F_OK    0       /*  Test for existence of file  */
```

```
#define ACCESS_WR       0x0002
#define ACCESS_RD       0x0004

/* Symbolic constants for the lseek() function */

#define SEEK_SET    0           /* Seek relative to the start
                                   of file   */
#define SEEK_CUR    1           /* Seek relative to current
                                   position   */
#define SEEK_END    2           /* Seek relative to the end
                                   of the file */

/* Symbolic constants for stream I/O */

#define STDIN_FILENO    0
#define STDOUT_FILENO   1
#define STDERR_FILENO   2
#ifndef NO_EXT_KEYS     /* extensions enabled */
#define STDAUX_FILENO   3
#define STDPRN_FILENO   4
#endif

int  access(const char *__path,int __mode);
int  chmod(const char *__path,int __pmode);
int  chsize(int __handle,long __size);
int  close(int __handle);
int  creat(const char *__path,int __pmode);
int  dup(int __handle);
int  dup2(int __handle1,int __handle2);
int  eof(int __handle);
long filelength(int __handle);
int  isatty(int __handle);
int  lock(int __handle,unsigned long __offset,unsigned long
          __nbytes);
long lseek(int __handle,long __offset,int __origin);
int  open(const char *__path,int __oflag,...);
int  read(int __handle,void *__buf,unsigned int __len);
int  setmode(int __handle,int __mode);
int  sopen(const char *__path,int __oflag,int __shflag,...);
long tell(int __handle);
int  umask(int __permission);
int  unlink(const char *__path);
int  unlock(int __handle,unsigned long __offset,unsigned long
            __nbytes);
int  write(int __handle,void *__buf,unsigned int __len);
#ifndef _STAT_H_INCLUDED
```

```
 #include <sys/stat.h>
#endif

#define _IO_H_INCLUDED
#endif
```

ANSI HEADER FILE LIMITS.H

```
/*
 *  limits.h    Machine and OS limits
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _LIMITS_H_INCLUDED

#define CHAR_BIT        8
#ifdef __CHAR_SIGNED__
 #define CHAR_MIN        (-128)
 #define CHAR_MAX        127
#else
 #define CHAR_MIN        0
 #define CHAR_MAX        255
#endif
#define MB_LEN_MAX      2
#define SCHAR_MAX       127
#define SCHAR_MIN       (-128)
#define UCHAR_MAX       255U
#define SHRT_MAX        32767
#define SHRT_MIN        (-32767-1)
#define USHRT_MAX       65535U
#ifdef __386__
 #define INT_MAX         2147483647
 #define INT_MIN         (-2147483647-1)
 #define UINT_MAX        4294967295U
#else
 #define INT_MAX         32767
 #define INT_MIN         (-32767-1)
 #define UINT_MAX        65535U
#endif
#define LONG_MAX        2147483647
#define LONG_MIN        (-2147483647-1)
#define ULONG_MAX       4294967295U
```

```
#define ACCESS_WR       0x0002
#define ACCESS_RD       0x0004

/* Symbolic constants for the lseek() function */

#define SEEK_SET    0          /* Seek relative to the start
                                  of file   */
#define SEEK_CUR    1          /* Seek relative to current
                                  position    */
#define SEEK_END    2          /* Seek relative to the end
                                  of the file */

/* Symbolic constants for stream I/O */

#define STDIN_FILENO    0
#define STDOUT_FILENO   1
#define STDERR_FILENO   2
#ifndef NO_EXT_KEYS     /* extensions enabled */
#define STDAUX_FILENO   3
#define STDPRN_FILENO   4
#endif

int  access(const char *__path,int __mode);
int  chmod(const char *__path,int __pmode);
int  chsize(int __handle,long __size);
int  close(int __handle);
int  creat(const char *__path,int __pmode);
int  dup(int __handle);
int  dup2(int __handle1,int __handle2);
int  eof(int __handle);
long filelength(int __handle);
int  isatty(int __handle);
int  lock(int __handle,unsigned long __offset,unsigned long
          __nbytes);
long lseek(int __handle,long __offset,int __origin);
int  open(const char *__path,int __oflag,...);
int  read(int __handle,void *__buf,unsigned int __len);
int  setmode(int __handle,int __mode);
int  sopen(const char *__path,int __oflag,int __shflag,...);
long tell(int __handle);
int  umask(int __permission);
int  unlink(const char *__path);
int  unlock(int __handle,unsigned long __offset,unsigned long
            __nbytes);
int  write(int __handle,void *__buf,unsigned int __len);
#ifndef _STAT_H_INCLUDED
```

```
 #include <sys/stat.h>
#endif

#define _IO_H_INCLUDED
#endif
```

ANSI HEADER FILE LIMITS.H

```
/*
 *  limits.h    Machine and OS limits
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _LIMITS_H_INCLUDED

#define CHAR_BIT        8
#ifdef __CHAR_SIGNED__
 #define CHAR_MIN        (-128)
 #define CHAR_MAX        127
#else
 #define CHAR_MIN        0
 #define CHAR_MAX        255
#endif
#define MB_LEN_MAX      2
#define SCHAR_MAX       127
#define SCHAR_MIN       (-128)
#define UCHAR_MAX       255U
#define SHRT_MAX        32767
#define SHRT_MIN        (-32767-1)
#define USHRT_MAX       65535U
#ifdef __386__
 #define INT_MAX         2147483647
 #define INT_MIN         (-2147483647-1)
 #define UINT_MAX        4294967295U
#else
 #define INT_MAX         32767
 #define INT_MIN         (-32767-1)
 #define UINT_MAX        65535U
#endif
#define LONG_MAX        2147483647
#define LONG_MIN        (-2147483647-1)
#define ULONG_MAX       4294967295U
```

```
#define TZNAME_MAX  30        /*  The maximum number of
                                  bytes                    */
                              /*  supported for the name of
                                  a time                   */
                              /*  zone (not of the TZ
                                  variable).               */

#define _LIMITS_H_INCLUDED
#endif
```

ANSI HEADER FILE LOCALE.H

```
/*
 *  locale.h
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _LOCALE_H_INCLUDED
#pragma pack(1);

#define LC_ALL      0
#define LC_COLLATE  1
#define LC_CTYPE    2
#define LC_NUMERIC  3
#define LC_TIME     4
#define LC_MONETARY 5

struct lconv
{
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
```

```
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
};

#if defined(__SMALL__) || defined(__MEDIUM__) || defined
                                                (__386__)
 #define NULL   0
#else
 #define NULL   0L
#endif

extern, char *setlocale(int __category,const char *__locale);
extern, struct lconv *localeconv(void);
#pragma pack();
#define _LOCALE_H_INCLUDED
#endif
```

EXTENSION HEADER FILE MALLOC.H

```
/*
 *  malloc.h, Memory allocation functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _MALLOC_H_INCLUDED
#pragma pack(1);

#ifndef _SIZE_T_DEFINED_
#define _SIZE_T_DEFINED_
typedef unsigned size_t;
#endif

#if defined(__SMALL__) || defined(__MEDIUM__) || defined
                                                 (__386__)
 #define NULL  0
#else
 #define NULL  0L
#endif

void  *alloca(size_t __size);
void  *_alloca(size_t __size);
```

```
#define __ALLOCA_ALIGN( s )   (((s)+(sizeof(int)-1))
                              &~(sizeof(int)-1))

#define alloca(size) \
( (__ALLOCA_ALIGN(size) < stackavail()) ? _alloca
    (__ALLOCA_ALIGN(size)) : NULL)

#if defined(__386__)
#pragma aux   _alloca = 0x29 0xc4 /* sub esp,eax */\
                        0x89 0xe0 /* mov eax,esp */\
                        parm caller [eax] value[eax]
                        modify [esp];
#elif defined(__SMALL__) || defined(__MEDIUM__) /* small
                        data models */
#pragma aux   _alloca = 0x29 0xc4 /* sub sp,ax */\
                        0x89 0xe0 /* mov ax,sp */\
                        parm caller [ax] value[ax] modify
                        [sp];
#else  /* large data models */
#pragma aux   _alloca = 0x29 0xc4 /* sub sp,ax */\
                        0x89 0xe0 /* mov ax,sp */\
                        0x8c 0xd2 /* mov dx,ss */\
                        parm caller [ax] value[ax dx] modify
                        [sp];
#endif

#define _HEAPOK         0
#define _HEAPEMPTY      1  /* heap isn't initialized */
#define _HEAPBADBEGIN 2  /* heap header is corrupted */
#define _HEAPBADNODE    3  /* heap entry is corrupted */
#define _HEAPEND        4  /* end of heap entries (_heapwalk) */
#define _HEAPBADPTR     5  /* invalid heap entry pointer
                              (_heapwalk) */

#define _USEDENTRY   0
#define _FREEENTRY   1

struct _heapinfo {
    void __far *_pentry;   /* heap pointer */
    size_t      _size;     /* heap entry size */
    int         _useflag;  /* heap entry 'in-use' flag */
};

int _heapchk( void );
int _nheapchk( void );
int _fheapchk( void );
```

```
int _heapset( unsigned int __fill );
int _nheapset( unsigned int __fill );
int _fheapset( unsigned int __fill );
int _heapwalk( struct _heapinfo *__entry );
int _nheapwalk( struct _heapinfo *__entry );
int _fheapwalk( struct _heapinfo *__entry );

void _heapgrow( void );
void _nheapgrow( void );
void _fheapgrow( void );
int _heapmin( void );
int _nheapmin( void );
int _fheapmin( void );
int _heapshrink( void );
int _nheapshrink( void );
int _fheapshrink( void );

int __nmemneed( size_t );
int __fmemneed( size_t );

void *calloc(size_t __n,size_t __size);
void *_expand(void *__ptr,size_t __size);
void __far  *_fexpand(void __far *__ptr,size_t __size);
void __near *_nexpand(void __near *__ptr,size_t __size);
void _ffree(void __far *__ptr);
void __far *_fmalloc(size_t __size);
void free(void *__ptr);
unsigned int _freect(size_t __size);
void __huge *halloc(long __n,size_t __size);
void hfree(void __huge *);
void *malloc(size_t __size);
void _nfree(void __near *__ptr);
void __near *_nmalloc(size_t __size);
void *realloc(void *__ptr,size_t __size);
void __near *_nrealloc(void __near *__ptr,size_t __size);
void __far  *_frealloc(void __far *__ptr,size_t __size);
size_t _msize( void *__ptr );
size_t _nmsize( void __near *__ptr );
size_t _fmsize( void __far *__ptr );
size_t _memavl(void);
size_t _memmax(void);
unsigned stackavail(void);

#pragma pack();
#define _MALLOC_H_INCLUDED
#endif
```

ANSI HEADER FILE MATH.H

```
/*
 *  math.h              Math functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _MATH_H_INCLUDED
#pragma pack(1);
extern double  _HugeValue;
#define HUGE_VAL   _HugeValue

double acos( double __x );
double asin( double __x );
double atan( double __x );
double atan2( double __y, double __x );
double ceil( double __x );
double cos( double __x );
double cosh( double __x );
double exp( double __x );
double fabs( double __x );
double floor( double __x );
double fmod( double __x, double __y );
double frexp( double __value, int *__exp );
double ldexp( double __x, int __exp );
double log( double __x );
double log10( double __x );
double modf( double __value, double *__iptr );
double pow( double __x, double __y );
double sin( double __x );
double sinh( double __x );
double sqrt( double __x );
double tan( double __x );
double tanh( double __x );

/* non-ANSI */
#ifndef NO_EXT_KEYS  /* extensions enabled */

struct, complex {
   double real;
   double imag;
};
```

```
double acosh( double __x );
double asinh( double __x );
double atanh( double __x );
double cabs( struct complex );
double hypot( double __x, double __y );
double j0( double __x );
double j1( double __x );
double jn( int __n, double __x );
double log2( double __x );
double y0( double __x );
double y1( double __x );
double yn( int __n, double __x );

/* The following struct is used to record errors detected in
 * the math library. matherr is called with a pointer to this
 * struct for possible error recovery.
 */

struct exception {
    int type;          /* type of error, see below */
    char *name;        /* name of math function */
    double arg1;       /* value of first argument to function */
    double arg2;       /* second argument (if indicated) */
    double retval;     /* default return value */
};

#define DOMAIN     1  /* argument domain error */
#define SING       2  /* argument singularity  */
#define OVERFLOW   3  /* overflow range error  */
#define UNDERFLOW  4  /* underflow range error */
#define TLOSS      5  /* total loss of significance */
#define PLOSS      6  /* partial loss of significance */

int    matherr( struct exception * );
double _matherr( struct exception * );
#endif, /* NO_EXT_KEYS */


#ifndef __NO_MATH_OPS
/*
    Defining the __NO_MATH_OPS macro will stop the compiler
    from recogizing the following functions as intrinsic
    operators.
*/
```

```
double __LOG( double __x );
double __COS( double __x );
double __SIN( double __x );
double __TAN( double __x );
double __SQRT( double __x );
double __FABS( double __x );
double __POW( double __x, double __y );
double __ATAN2( double __y, double __x );
double __FMOD( double __x, double __y );
double __ACOS( double __x );
double __ASIN( double __x );
double __ATAN( double __x );
double __COSH( double __x );
double __EXP( double __x );
double __LOG10( double __x );
double __SINH( double __x );
double __TANH( double __x );

#define log( __x )            __LOG( __x )
#define cos( __x )            __COS( __x )
#define sin( __x )            __SIN( __x )
#define tan( __x )            __TAN( __x )
#define sqrt( __x )           __SQRT( __x )
#define fabs( __x )           __FABS( __x )
#define pow( __x, __y )       __POW( __x, __y )
#define atan2( __y, __x )     __ATAN2( __y, __x )
#define fmod( __x, __y )      __FMOD( __x, __y )
#define acos( __x )           __ACOS( __x )
#define asin( __x )           __ASIN( __x )
#define atan( __x )           __ATAN( __x )
#define cosh( __x )           __COSH( __x )
#define exp( __x )            __EXP( __x )
#define log10( __x )          __LOG10( __x )
#define sinh( __x )           __SINH( __x )
#define tanh( __x )           __TANH( __x )

#endif
#pragma pack();
#define _MATH_H_INCLUDED
#endif
```

EXTENSION HEADER FILE PROCESS.H

```
/*
 *  process.h      Process spawning and related routines
```

```
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _PROCESS_H_INCLUDED

/* mode flags for spawnxxx routines */
extern int __p_overlay;

#define P_WAIT      0
#define P_NOWAIT    1
#define P_OVERLAY   __p_overlay
#define P_NOWAITO   3

/* values for __action_code used with cwait() */

#define WAIT_CHILD      0
#define WAIT_GRANDCHILD 1

void  abort(void);
int __far _beginthread( void (__far *__start_address)(void
    __far *), void __far *__stack_bottom, unsigned
    __stack_size,void __far *__arglist );
int   cwait(int *__status,int __process_id,int __action_code);
void __far _endthread();
#ifndef __386__
int   execl(const char *__path, const char *__arg0,...);
int   execle(const char *__path, const char *__arg0,...);
int   execlp(const char *__path, const char *__arg0,...);
int   execlpe(const char *__path, const char *__arg0,...);
int   execv(const char *__path,char **__argv);
int   execve(const char *__path,char **__argv,char **__envp);
int   execvp(const char *__path,char **__argv);
int   execvpe(const char *__path,char **__argv,char **__envp);
#endif
void  exit(int __status);
void  _exit(int __status);
char *getcmd(char *__buffer);
char *getenv(const char *__name);
int   getpid(void);
int   putenv(const char *__string);
int   spawnl(int __mode,const char *__path, const char
      *__arg0,...);
int   spawnle(int __mode,const char *__path, const char
      *__arg0,...);
```

```
int     spawnlp(int __mode,const char *__path, const char
        *__arg0,...);
int     spawnlpe(int __mode,const char *__path, const char
        *__arg0,...);
int     spawnv(int __mode,const char *__path,char **__argv);
int     spawnve(int __mode,const char *__path,char
        **__argv,char **__envp);
int     spawnvp(int __mode,const char *__path,char **__argv);
int     spawnvpe(int __mode,const char *__path,char
        **__argv,char **__envp);
int     system(const char *__cmd);
int     wait(int *__status);

#pragma aux abort  aborts;
#pragma aux exit   aborts;
#pragma aux _exit  aborts;

#define _PROCESS_H_INCLUDED
#endif
```

EXTENSION HEADER FILE SEARCH.H

```
/*
 *  search.h       Function prototypes for searching functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1990-1991. All
    rights reserved.
 */
#ifndef _SEARCH_H_INCLUDED

void *lfind(const void *__key, const void *__base, unsigned
        *__num, unsigned __width, int (* __compare)(const
        void *, const void *));
void *lsearch(const void *__key, const void *__base, unsigned
        *__num, unsigned __width, int (* __compare)(const
        void *, const void *));
#define _SEARCH_H_INCLUDED
#endif
```

ANSI HEADER FILE SETJMP.H

```
/*
 *  setjmp.h
 *
```

```
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _SETJMP_H_INCLUDED

typedef unsigned int jmp_buf[13];

#define setjmp(__env)     _setjmp(__env)
#if defined(__386__)
 #pragma aux _setjmp parm caller [eax] modify [8087];
#else
 #pragma aux _setjmp modify [8087];
#endif

int  _setjmp( jmp_buf __env );
void longjmp( jmp_buf __env, int __val );

#pragma aux  longjmp aborts;

#define _SETJMP_H_INCLUDED
#endif
```

EXTENSION HEADER FILE SHARE.H

```
/*
 *  share.h    Define file sharing modes for sopen()
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */

#define SH_COMPAT   0x00    /* compatibility mode   */
#define SH_DENYRW   0x10    /* deny read/write mode */
#define SH_DENYWR   0x20    /* deny write mode,     */
#define SH_DENYRD   0x30    /* deny read mode,      */
#define SH_DENYNO   0x40    /* deny none mode,      */
```

ANSI HEADER FILE SIGNAL.H

```
/*
 *  signal.h
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
```

```
 */
#ifndef _SIGNAL_H_INCLUDED

typedef int    sig_atomic_t;

#define SIG_IGN      (void (*)(int)) 1
#define SIG_DFL      (void (*)(int)) 2
#define SIG_ERR      (void (*)(int)) 3

#define SIGABRT   1
#define SIGFPE    2
#define SIGILL    3
#define SIGINT    4
#define SIGSEGV   5
#define SIGTERM   6
#define SIGBREAK  7
/* following are OS/2 process flag A,B and C */
#define SIGUSR1   8
#define SIGUSR2   9
#define SIGUSR3  10

#define _SIGMIN   1
#define _SIGMAX  10

void (*signal( int __sig, void (*__func)() ) )();
int  raise( int __sig );

#define _SIGNAL_H_INCLUDED
#endif
```

ANSI HEADER FILE STDARG.H

```
/*
 *  stdarg.h       Variable argument macros
 *                 definitions for use with variable argument
 *                 lists
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _STDARG_H_INCLUDED

#ifdef  __HUGE__
typedef char __far *va_list[1];
#else
```

```
typedef char *va_list[1];
#endif

#ifdef  __HUGE__
#define va_start(ap,pn)  ((ap)[0]=(char __far*)&pn+((sizeof
    (pn)+1)&~1),(void)0)
#define va_arg(ap,type)  ((ap)[0]+=((sizeof(type)+1)&~1),\
    (*(type __far *)((ap)[0]-((sizeof(type)+1)&~1))))
#define va_end(ap),  ((ap)[0]=0,(void)0)
#else
#define va_start(ap,pn)  ((ap)[0]=(char *)&pn+\
    ((sizeof(pn)+sizeof(int)-1)&~(sizeof(int)-1)),(void)0)
#define va_arg(ap,type)     ((ap)[0]+=\
    ((sizeof(type)+sizeof(int)-1)&~(sizeof(int)-1)),\
    (*(type *)((ap)[0]-((sizeof(type)+sizeof(int)-1)&~
    (sizeof(int)-1)))))
#define va_end(ap),     ((ap)[0]=0,(void)0)
#endif

#define _STDARG_H_INCLUDED
#endif
```

ANSI HEADER FILE STDDEF.H

```
/*
 *  stddef.h        Standard definitions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _STDDEF_H_INCLUDED

#ifndef _SIZE_T_DEFINED_
#define _SIZE_T_DEFINED_
typedef unsigned size_t;
#endif

#ifndef _WCHAR_T_DEFINED_
#define _WCHAR_T_DEFINED_
typedef unsigned short wchar_t;
#endif

#if defined(__SMALL__) || defined(__MEDIUM__)
                       || defined(__386__)
 #define NULL   0
```

```
#else
 #define NULL   0L
#endif

#if defined(__HUGE__)
 typedef long   ptrdiff_t;
#else
 typedef int    ptrdiff_t;
#endif

#define offsetof(typ,id) (size_t)&(((typ*)0)->id)

/* define pragma for 'cdecl'  keyword to match Microsoft
   conventions */
/* define pragma for 'pascal' keyword to match Microsoft
   conventions */
#if defined( __WINDOWS_386__ )
 #pragma aux cdecl "_*" parm caller []\
   value struct float struct routine [eax]  modify [eax ebx
   ecx edx fs gs];
 #pragma aux pascal "^" parm routine reverse []\
   value struct float struct routine [eax]  modify [eax ebx
   ecx edx fs gs];
 #pragma aux fortran "^";
#elif defined( __386__ )
 #pragma aux cdecl "_*" parm caller []\
   value struct float struct routine [eax]  modify [eax ebx
   ecx edx];
 #pragma aux pascal "^" parm routine reverse []\
   value struct float struct routine [eax]  modify [eax ebx
   ecx edx];
 #pragma aux fortran "^";
#else
 #pragma aux cdecl "_*" parm caller loadds []\
   value struct float struct routine [ax]  modify [ax bx cx
   dx es];
 #pragma aux pascal "^" parm routine reverse []\
   value struct float struct routine [ax]  modify [ax bx cx
   dx es];
 #pragma aux fortran "^";
#endif
#ifndef NO_EXT_KEYS, /* extensions enabled */
 extern int __far *_threadid;   /* pointer to thread id */
#endif
```

```
#define _STDDEF_H_INCLUDED
#endif
```

ANSI HEADER FILE STDIO.H

```
/*
 *  stdio.h     Standard I/O functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _STDIO_H_INCLUDED
#pragma pack(1);

#ifndef _SIZE_T_DEFINED_
 #define _SIZE_T_DEFINED_
 typedef unsigned size_t;
#endif

#if defined(__SMALL__) || defined(__MEDIUM__) || defined
          (__386__)
 #define NULL   0
#else
 #define NULL   0L
#endif

#ifdef  __HUGE__
 typedef char __far *__va_list[1];
#else
 typedef char *__va_list[1];
#endif

#ifdef __386__
 #define BUFSIZ         4096
#else
 #define BUFSIZ         512
#endif
#define _NFILES         20      /* number of files that can
                                   be handled */
#define FILENAME_MAX    80

typedef struct  __iobuf {
    char            *_ptr;          /* next character
                                       position */
    int             _cnt;           /* number of characters
```

```
                                        left */
    char            *_base;          /* location of buffer */
    unsigned        _flag;           /* mode of file access */
    int             _handle;         /* file handle */
    unsigned        _bufsize;        /* size of buffer */
    unsigned char   _ungotten;       /* character placed here
                                        by ungetc */
    unsigned char   _tmpfchar;       /* tmpfile number */
} FILE;

typedef long    fpos_t;

#ifndef NO_EXT_KEYS     /* extensions enabled */
 #define FOPEN_MAX      _NFILES
 #define OPEN_MAX       FOPEN_MAX
 #define PATH_MAX       80
#else                   /* extensions not enabled */
 #define FOPEN_MAX      (_NFILES-2)
#endif

extern  FILE    __near __iob[_NFILES];
extern  unsigned __near _iomode[_NFILES];
#define stdin   ((FILE *)&__iob[0])     /* standard input
                                           file  */
#define stdout  ((FILE *)&__iob[1])     /* standard output
                                           file */
#define stderr  ((FILE *)&__iob[2])     /* standard error
                                           file  */
#ifndef NO_EXT_KEYS     /* extensions enabled */
#define stdaux  ((FILE *)&__iob[3])     /* standard auxiliary
                                           file  */
#define stdprn  ((FILE *)&__iob[4])     /* standard printer
                                           file  */
#endif

/* values for _flag field in FILE struct and _iomode array */

#define _READ   0x0001  /* file opened for reading */
#define _WRITE  0x0002  /* file opened for writing */
#define _UNGET  0x0004  /* ungetc has been done */
#define _BIGBUF 0x0008  /* big buffer allocated */
#define _EOF    0x0010  /* EOF has occurred */
#define _ERR    0x0020  /* error has occurred on this file */
#define _BINARY 0x0040  /* file is binary, skip CRLF
                           processing */
#define _APPEND 0x0080  /* file opened for append */
```

```
#define _IOFBF  0x0100  /* full buffering */
#define _IOLBF  0x0200  /* line buffering */
#define _IONBF  0x0400  /* no buffering */
#define _TMPFIL 0x0800  /* this is a temporary file */
#define _DIRTY  0x1000  /* buffer has been modified */
#define _ISTTY  0x2000  /* is console device */

#define EOF     (-1)

#define SEEK_SET        0
#define SEEK_CUR        1
#define SEEK_END        2

#define L_tmpnam        13
#define TMP_MAX         (26*26*26)

int    _bprintf(char *__buf,unsigned int __bufsize,const char
                *__format,...);
void   clearerr(FILE *__fp);
int    fclose(FILE *__fp);
int    fcloseall(void);
FILE *fdopen(int __handle,const char *__mode);
int    feof(FILE *__fp);
int    ferror(FILE *__fp);
int    fflush(FILE *__fp);
int    fgetc(FILE *__fp);
int    fgetchar(void);
int    fgetpos(FILE *__fp,fpos_t *__pos);
char *fgets(char *__s,int __n,FILE *__fp);
int    flushall(void);
FILE *fopen(const char *__filename,const char *__mode);
int    fprintf(FILE *__fp,const char *__format,...);
int    fputc(int __c,FILE *__fp);
int    fputchar(int __c);
int    fputs(const char *__s,FILE *__fp);
size_t fread(void *__ptr,size_t __size,size_t __n,FILE *__fp);
FILE *freopen(const char *__filename,const char *__mode,FILE
              *__fp);
int    fscanf(FILE*__fp,const char *__format,...);
int    fseek(FILE *__fp,long int __offset,int __whence);
int    fsetpos(FILE *__fp,const fpos_t *__pos);
long int ftell(FILE *__fp);
size_t fwrite(const void *__ptr,size_t __size,size_t __n,
             FILE *__fp);
int    getc(FILE *__fp);
int    getchar(void);
```

```
char *gets(char *__s);
void  perror(const char *__s);
int   printf(const char *__format,...);
int   putc(int __c,FILE *__fp);
int   putchar(int __c);
int   puts(const char *__s);
int   remove(const char *__filename);
int   rename(const char *__old,const char *__new);
void  rewind(FILE *__fp);
int   scanf(const char *__format,...);
void  setbuf(FILE *__fp,char *__buf);
int   setvbuf(FILE *__fp,char *__buf,int __mode,size_t
              __size);
int   sprintf(char *__s,const char *__format,...);
int   sscanf(const char *__s,const char *__format,...);
FILE *tmpfile(void);
char *tmpnam(char *__s);
int   _vbprintf(char *__buf,unsigned int __bufsize,const
                char *__format, __va_list __arg);
int   vfprintf(FILE *__fp,const char *__format,__va_list
               __arg);
int   vprintf(const char *__format,__va_list __arg);
int   vsprintf(char *__s,const char *__format,__va_list
               __arg);
int   vfscanf(FILE *__fp,const char *__format,__va_list
              __arg);
int   vscanf(const char *__format,__va_list __arg);
int   vsscanf(const char *__s,const char *__format,__va_list
              __arg);
int   ungetc(int __c,FILE *__fp);

#define clearerr(fp)    ((fp)->_flag &= ~(_ERR|_EOF))
#define feof(fp)        ((fp)->_flag & _EOF)
#define ferror(fp)      ((fp)->_flag & _ERR)
#define fileno(fp)      ((fp)->_handle)
#define getchar()       fgetc(stdin)
#define getc(fp)        fgetc((fp))
#define putchar(c)      fputc((c),stdout)
#define putc(c,fp)      fputc((c),(fp))

#pragma pack();
#define _STDIO_H_INCLUDED
#endif
```

ANSI HEADER FILE STDLIB.H

```
/*
 *  stdlib.h    Standard Library functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _STDLIB_H_INCLUDED
#pragma pack(1);

#ifndef _SIZE_T_DEFINED_
 #define _SIZE_T_DEFINED_
 typedef unsigned size_t;
#endif

#ifndef _WCHAR_T_DEFINED_
 #define _WCHAR_T_DEFINED_
 typedef unsigned short wchar_t;
#endif

#if defined(__SMALL__) || defined(__MEDIUM__) || defined
                                                  (__386__)
 #define NULL   0
#else
 #define NULL   0L
#endif


#define RAND_MAX        32767u
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    0xff
#define MB_CUR_MAX      1

typedef struct
{
   int     quot;
   int     rem;
} div_t;

typedef struct
{
   long    quot;
   long    rem;
} ldiv_t;
```

```
/* min and max macros */

#define max(a,b)   (((a) > (b)) ? (a) : (b))
#define min(a,b)   (((a) < (b)) ? (a) : (b))

/*
 * The following sizes are the maximum sizes of buffers used
 * by the _makepath() and _splitpath() functions. They
 * include space for the '\0' terminator.
 */

#define _MAX_PATH    144 /* maximum length of full pathname */
#define _MAX_DRIVE     3 /* maximum length of drive component */
#define _MAX_DIR     130 /* maximum length of path component */
#define _MAX_FNAME     9 /* maximum length of file name
                            component */
#define _MAX_EXT       5 /* maximum length of extension
                            component */

#ifdef  __INLINE_FUNCTIONS__
 int      abs(int __j);
 div_t    div(int __numer,int __denom);
 long int labs(long int __j);
 unsigned int _rotl(unsigned int __value,unsigned int __shift);
 unsigned int _rotr(unsigned int __value,unsigned int __shift);
 #ifdef __386__
  unsigned long _lrotl(unsigned long __value,unsigned int
                __shift);
  unsigned long _lrotr(unsigned long __value,unsigned int
                __shift);
  #define _lrotl(x,n) _inline__lrotl(x,n)
  #define _lrotr(x,n) _inline__lrotr(x,n)
 #endif
 #define abs(x)     _inline_abs(x)
 #define div(x,y)   _inline_div(x,y)
 #define labs(x)    _inline_labs(x)
 #define _rotl(x,n) _inline__rotl(x,n)
 #define _rotr(x,n) _inline__rotr(x,n)
#endif

#pragma aux    abort   aborts;
#pragma aux    exit    aborts;
#pragma aux    _exit   aborts;

void     abort(void);
int      abs(int __j);
```

```
int       atexit(void (*__func)(void));
double    atof(const char *__nptr);
int       atoi(const char *__nptr);
long int atol(const char *__nptr);
void     *bsearch(const void *__key,const void *__base,
                  size_t __nmemb,size_t __size,
                  int (*__compar)(const void *__pkey,const
                  void *__pbase));
void     *calloc(size_t __n,size_t __size);
div_t     div(int __numer,int __denom);
char     *ecvt(double __val,int __ndig,int *__dec,int *__sign);
void      exit(int __status);
void      _exit(int __status);
char     *fcvt(double __val,int __ndig,int *__dec,int *__sign);
void      free(void *__ptr);
char     *_fullpath(char *__buf,const char *__path,size_t
                    __size);
char     *gcvt(double __val,int __ndig,char *__buf);
char     *getenv(const char *__name);
char     *itoa(int __value,char *__buf,int __radix);
long int labs(long int __j);
ldiv_t    ldiv(long int __numer,long int __denom);
unsigned long _lrotl(unsigned long __value,unsigned int
                     __shift);
unsigned long _lrotr(unsigned long __value,unsigned int
                     __shift);
char     *ltoa(long int __value,char *__buf,int __radix);
void      _makepath(char *__path,const char *__drive,const
                    char *__dir, const char *__fname,const
                    char *__ext);
void     *malloc(size_t __size);
int       mblen(const char *__s,size_t __n);
size_t    mbstowcs(wchar_t *__pwcs,const char *__s,size_t __n);
int       mbtowc(wchar_t *__pwc,const char *__s,size_t __n);
int       putenv(const char *__string);
void      qsort(void *__base,size_t __nmemb,size_t __size,
               int (*__compar)(const void *,const void *));
int       rand(void);
void     *realloc(void *__ptr,size_t __size);
unsigned int _rotl(unsigned int __value,unsigned int __shift);
unsigned int _rotr(unsigned int __value,unsigned int __shift);
void      _searchenv(const char *__name,const char *__env_var,
                     char *__buf);
void      _splitpath(const char *__path,char *__drive,
                     char *__dir,char *__fname,char *__ext);
```

```
void      srand(unsigned int __seed);
double    strtod(const char *__nptr,char **__endptr);
long int strtol(const char *__nptr,char **__endptr,int
                __base);
unsigned long int strtoul(const char *__nptr,char **__endptr,
                          int __base);
void      swab( char *__src, char *__dest, int __num );
int       system(const char *__string);
char     *utoa(unsigned int __value,char *__buf,int __radix);
char     *ultoa(unsigned long int __value,char *__buf,int
              __radix);
size_t    wcstombs(char *__s,const wchar_t *__pwcs,size_t __n);
int       wctomb(char *__s,wchar_t __wchar);

#define atof(p) strtod(p,(char **)NULL)

/* external variables */
#ifndef NO_EXT_KEYS     /* extensions enabled */
 void __near *__brk(unsigned __new_brk_value);
 void __near *sbrk(int __increment);
 typedef void (*onexit_t)();
 onexit_t onexit(onexit_t __func);
 extern  char **  __near environ;     /* pointer to
                                         environment table */
#endif

#define errno (*__get_errno_ptr())
#define _doserrno (*__get_doserrno_ptr())
extern  volatile int    errno;          /* ANSI error
                                           number */
extern  int             _doserrno;      /* DOS system error
                                           code value */
extern  unsigned __near _psp;           /* Program Segment
                                           Prefix */
extern  unsigned char __near _osmajor;  /* DOS version # */
extern  unsigned char __near _osminor;  /* DOS version # */
#define DOS_MODE 0                      /* Real Address
                                           Mode */
#define OS2_MODE 1                      /* Protected Address
                                           Mode */
extern  unsigned char __near _osmode;   /* DOS_MODE or
                                           OS2_MODE */
extern  int       __near  _fmode;       /* default file
                                           translation mode */
extern unsigned __near  _amblksiz;      /* DOS mallocs done
                                           in multiples of */
```

```
extern char *            sys_errlist[]; /* strerror error
                                          message table */
extern int      __near  sys_nerr;       /* # of entries on
                                          sys_errlist
                                          array */
extern unsigned __near, __minreal;    /* DOS4GW var for WLINK
                                          MINREAL option */

#pragma pack();
#define _STDLIB_H_INCLUDED
#endif
```

ANSI HEADER FILE STRING.H

```
/*
 *  string.h        String functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _STRING_H_INCLUDED

#ifndef _SIZE_T_DEFINED_
#define _SIZE_T_DEFINED_
typedef unsigned size_t;
#endif

#if defined(__SMALL__) || defined(__MEDIUM__) || defined
                                                (__386__)
 #define NULL   0
#else
 #define NULL   0L
#endif

#ifdef   __INLINE_FUNCTIONS__
 void *memchr( const void *__s, int __c, size_t __n );
 int   memcmp( const void *__s1, const void *__s2, size_t __n );
 void *memcpy( void *__s1, const void *__s2, size_t __n );
 void *memmove( void *__s1, const void *__s2, size_t __n );
 void *memset( void *__s, int __c, size_t __n );
 char *strcat( char *__s1, const char *__s2 );
 char *strchr( const char *__s, int __c );
 int   strcmp( const char *__s1, const char *__s2 );
 char *strcpy( char *__s1, const char *__s2 );
 size_t strlen( const char *__s );
```

```
#ifndef NO_EXT_KEYS
/* extensions enabled */
 void __far *_fmemchr( const void __far *__s, int __c, size_t
                       __n );
 void __far *_fmemcpy( void __far *__s1, const void __far
                       *__s2, size_t __n );
 void __far *_fmemset( void __far *__s, int __c, size_t __n );
 int  _fmemcmp( const void __far *__s1, const void __far
                *__s2, size_t __n );
 char __far *_fstrcat( char __far *__s1, const char __far
                       *__s2 );
 char __far *_fstrchr( const char __far *__s, int __c );
 int  _fstrcmp( const char __far *__s1, const char __far
                *__s2 );
 char __far *_fstrcpy( char __far *__s1, const char __far
                       *__s2 );
 size_t _fstrlen( const char __far *__s );
 void movedata( unsigned __srcseg, unsigned __srcoff,
 unsigned __tgtseg, unsigned __tgtoff, unsigned __len );
 #define movedata(ds,si,es,di,cx) _inline_movedata
                (ds,si,es,di,cx)
 #define _fmemchr(p,c,n)       _inline__fmemchr(p,c,n)
 #define _fmemcmp(p1,p2,n)     _inline__fmemcmp(p1,p2,n)
 #define _fmemcpy(p1,p2,n)     _inline__fmemcpy(p1,p2,n)
 #define _fmemset(p,c,n)       _inline__fmemset(p,c,n)
 #define _fstrcat(p1,p2)       _inline__fstrcat(p1,p2)
 #define _fstrcmp(p1,p2)       _inline__fstrcmp(p1,p2)
 #define _fstrcpy(p1,p2)       _inline__fstrcpy(p1,p2)
 #define _fstrlen(p1)          _inline__fstrlen(p1)
#endif
 #define memchr(p,c,n)         _inline_memchr(p,c,n)
 #define memcmp(p1,p2,n)       _inline_memcmp(p1,p2,n)
 #define memcpy(p1,p2,n)       _inline_memcpy(p1,p2,n)
 #define memset(p,c,n)         _inline_memset(p,c,n)
 #define strcat(p1,p2)         _inline_strcat(p1,p2)
 #define strcmp(p1,p2)         _inline_strcmp(p1,p2)
 #define strcpy(p1,p2)         _inline_strcpy(p1,p2)
 #define strlen(p1)            _inline_strlen(p1)
 #define strchr(p1,p2)         _inline_strchr(p1,p2)
#endif

void *memchr( const void *__s, int __c, size_t __n );
int   memcmp( const void *__s1, const void *__s2, size_t __n );
void *memcpy( void *__s1, const void *__s2, size_t __n );
void *memmove( void *__s1, const void *__s2, size_t __n );
```

```
void *memset( void *__s, int __c, size_t __n );
char *strcat( char *__s1, const char *__s2 );
char *strchr( const char *__s, int __c );
int   strcmp( const char *__s1, const char *__s2 );
int   strcoll( const char *__s1, const char *__s2 );
char *strcpy( char *__s1, const char *__s2 );
size_t strcspn( const char *__s1, const char *__s2 );
char *strncat( char *__s1, const char *__s2, size_t __n );
char *strerror( int __errnum );
size_t strlen( const char *__s );
int   strncmp( const char *__s1, const char *__s2, size_t __n );
char *strncpy( char *__s1, const char *__s2, size_t __n );
char *strpbrk( const char *__s1, const char *__s2 );
char *strrchr( const char *__s, int __c );
size_t strspn( const char *__s1, const char *__s2 );
char *strstr( const char *__s1, const char *__s2 );
char *strtok( char *__s1, const char *__s2 );
size_t strxfrm( char *__s1, const char *__s2, size_t __n );

/* non-ANSI */
#ifndef NO_EXT_KEYS
/* extensions enabled */
void __far *_fmemccpy(void __far *__s1,const void __far
                      *__s2,int __c,size_t __n);
void __far *_fmemchr( const void __far *__s, int __c, size_t
                      __n );
void __far *_fmemcpy( void __far *__s1, const void __far
                      *__s2, size_t __n );
void __far *_fmemmove( void __far *__s1, const void __far
                       *__s2, size_t __n );
void __far *_fmemset( void __far *__s, int __c, size_t __n );
int   _fmemcmp( const void __far *__s1, const void __far
                *__s2, size_t __n );
int   _fmemicmp( const void __far *__s1, const void __far
                 *__s2, size_t __n );
char __far *_fstrcat( char __far *__s1, const char __far
                      *__s2 );
char __far *_fstrchr( const char __far *__s, int __c );
int   _fstrcmp( const char __far *__s1, const char __far *__s2 );
char __far *_fstrcpy( char __far *__s1, const char __far
                      *__s2 );
size_t _fstrcspn( const char __far *__s1, const char __far
                  *__s2 );
int   _fstricmp( const char __far *__s1, const char __far
                 *__s2 );
```

```
char __far *_fstrncat( char __far *__s1, const char __far
                        *__s2, size_t __n );
size_t _fstrlen( const char __far *__s );
char __far *_fstrlwr( char __far *__string );
int  _fstrncmp( const char __far *__s1, const char __far
                 *__s2, size_t __n );
char __far *_fstrncpy( char __far *__s1, const char __far
                        *__s2, size_t __n );
int  _fstrnicmp( const char __far *__s1, const char __far
                  *__s2, size_t __n );
char __far *_fstrnset( char __far *__string, int __c, size_t
                        __len );
char __far *_fstrpbrk( const char __far *__s1, const char
                        __far *__s2 );
char __far *_fstrrchr( const char __far *__s, int __c );
char __far *_fstrrev( char __far *__string );
char __far *_fstrset( char __far *__string, int __c );
size_t _fstrspn( const char __far *__s1, const char __far
                  *__s2 );
char __far *_fstrstr( const char __far *__s1, const char
                       __far *__s2 );
char __far *_fstrtok( char __far *__s1, const char __far
                       *__s2 );
char __far *_fstrupr( char __far *__string );
void *memccpy( void *__s1, const void *__s2, int __c, size_t
                __n );
int  memicmp( const void *__s1, const void *__s2, size_t __n );
void movedata( unsigned __srcseg, unsigned __srcoff,
       unsigned __tgtseg, unsigned __tgtoff, unsigned __len );
int   strcmpi( const char *__s1, const char *__s2 );
char *strdup( const char *__string );
int   stricmp( const char *__s1, const char *__s2 );
char *strlwr( char *__string );
int   strnicmp( const char *__s1, const char *__s2, size_t
                 __n );
char *strnset( char *__string, int __c, size_t __len );
char *strrev( char *__string );
char *strset( char *__string, int __c );
char *strupr( char *__string );
#endif

#define _STRING_H_INCLUDED
#endif
```

ANSI HEADER FILE TIME.H

```
/*
 *  time.h       Time functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _TIME_H_INCLUDED
#pragma pack(1);

#ifndef _SIZE_T_DEFINED_
 #define _SIZE_T_DEFINED_
 typedef unsigned size_t;
#endif

#if defined(__SMALL__) || defined(__MEDIUM__) || defined
                                                 (__386__)
 #define NULL   0
#else
 #define NULL   0L
#endif

#ifndef _TIME_T_DEFINED_
 #define _TIME_T_DEFINED_
 typedef unsigned long time_t;, /* time value */
#endif

#define CLK_TCK , 100
#define CLOCKS_PER_SEC, 100
typedef unsigned long, clock_t;

struct tm
{
   int  tm_sec;    /* seconds after the minute -- [0,61] */
   int  tm_min;    /* minutes after the hour   -- [0,59] */
   int  tm_hour;   /* hours after midnight     -- [0,23] */
   int  tm_mday;   /* day of the month         -- [1,31] */
   int  tm_mon;    /* months since January     -- [0,11] */
   int  tm_year;   /* years since 1900                   */
   int  tm_wday;   /* days since Sunday        -- [0,6]  */
   int  tm_yday;   /* days since January 1     -- [0,365]*/
   int  tm_isdst;, /* Daylight Savings Time flag */
};
```

```
char    *asctime( const struct tm *__timeptr );
clock_t clock(void);
char    *ctime( const time_t *__timer );
double difftime( time_t __t1, time_t __t0 );
struct tm *gmtime( const time_t *__timer );
struct tm *localtime( const time_t *__timer );
time_t mktime( struct tm *__timeptr );
size_t strftime(char *__s,size_t __maxsiz,const char *__fmt,
        const struct tm *__tp );
time_t time( time_t *__timer );

#define difftime(t1,t0)  ((double)(t1)-(double)(t0))

#ifndef NO_EXT_KEYS, /* extensions enabled */
 void   tzset(void);
 extern long  timezone;    /* # of seconds from GMT */
 extern int   daylight;    /* d.s.t. indicator */
 extern char  *tzname[2];  /* time zone names */
#endif

#pragma pack();
#define _TIME_H_INCLUDED
#endif
```

EXTENSION HEADER FILE VARARGS.H

```
/*
 *  varargs.h   Variable argument macros (UNIX System V
 *              definition) definitions for use with variable
 *              argument lists
 *
 *  Copyright (C) by WATCOM Systems Inc. 1991. All rights
    reserved.
 */
#ifndef _VARARGS_H_INCLUDED

#ifndef _STDARG_H_INCLUDED
 #include <stdarg.h>
#else
 #error stdarg.h has already been included
#endif

#define va_alist   void *__alist, ...
#define va_dcl
```

```
#undef va_start
#define va_start(ap)   ((ap)[0]=(void *)&__alist,(void)0)

#define _VARARGS_H_INCLUDED
#endif
```

EXTENSION HEADER FILE GRAPH.H

```
/*
 *  graph.h      Graphics functions
 *
 *  Copyright (C) by WATCOM Systems Inc. 1988-1991. All
    rights reserved.
 */
#ifndef _GRAPH_H_INCLUDED
#pragma pack(1);

#if defined ( __386__ )
    #define _FAR
    #define _HUGE
#else
    #define _FAR      __far
    #define _HUGE     __huge
#endif
#pragma library (graph);

struct xycoord {                /* structure for pixel
                                   position */

        short   xcoord;
        short   ycoord;
};

struct _wxycoord {              /* structure for window
                                   position*/

        double  wx;
        double  wy;
};

struct rccoord {                /* structure for text
                                   position  */

        short   row;
        short   col;
};
```

```
struct videoconfig {            /* structure for
                                   _getvideoconfig */
        short   numxpixels;
        short   numypixels;
        short   numtextcols;
        short   numtextrows;
        short   numcolors;
        short   bitsperpixel;
        short   numvideopages;
        short   mode;
        short   adapter;
        short   monitor;
        short   memory;
};

struct textsettings {           /* structure for
                                   _gettextsettings */
        short   basevectorx;
        short   basevectory;
        short   txpath;
        short   height;
        short   width;
        short   spacing;
        short   horizalign;
        short   vertalign;
};

/* Video Setup and Query Functions */

short _FAR               _setvideomode( short );
short _FAR               _setvideomoderows( short, short );
struct videoconfig _FAR * _FAR
                         _getvideoconfig( struct videoconfig
                                           _FAR * );
short _FAR               _grstatus( void );
short _FAR               _setactivepage( short );
short _FAR               _getactivepage( void );
short _FAR               _setvisualpage( short );
short _FAR               _getvisualpage( void );

#define _MAXRESMODE      (-3)    /* graphics mode with highest
                                    res.                     */
#define _MAXCOLORMODE    (-2)    /* graphics mode with most
                                    colors                   */
#define _DEFAULTMODE     (-1)    /* restore screen to original
                                    mode                     */
```

```
#define _TEXTBW40        0        /* 40 x 25 text, 16 gray    */
#define _TEXTC40         1        /* 40 x 25 text, 16/8 color */
#define _TEXTBW80        2        /* 80 x 25 text, 16 gray    */
#define _TEXTC80         3        /* 80 x 25 text, 16/8 color */
#define _MRES4COLOR      4        /* 320 x 200, 4 color       */
#define _MRESNOCOLOR     5        /* 320 x 200, 4 gray        */
#define _HRESBW          6        /* 640 x 200, BW            */
#define _TEXTMONO        7        /* 80 x 25 text, BW         */
#define _HERCMONO        11       /* 720 x 350, BW            */
#define _MRES16COLOR     13       /* 320 x 200, 16 color      */
#define _HRES16COLOR     14       /* 640 x 200, 16 color      */
#define _ERESNOCOLOR     15       /* 640 x 350, BW            */
#define _ERESCOLOR       16       /* 640 x 350, 4 or 16 color */
#define _VRES2COLOR      17       /* 640 x 480, BW            */
#define _VRES16COLOR     18       /* 640 x 480, 16 color      */
#define _MRES256COLOR    19       /* 320 x 200, 256 color     */
#define _URES256COLOR    0x100    /* 640 x 400, 256 color     */
#define _VRES256COLOR    0x101    /* 640 x 480, 256 color     */
#define _SVRES16COLOR    0x102    /* 800 x 600, 16 color      */
#define _SVRES256COLOR   0x103    /* 800 x 600, 256 color     */
#define _XRES16COLOR     0x104    /* 1024 x 768, 16 color     */
#define _XRES256COLOR    0x105    /* 1024 x 768, 256 color    */

#define _NODISPLAY       (-1)     /* no display device        */
#define _UNKNOWN         0        /* unknown adapter/monitor
                                     type                     */

#define _MDPA            1        /* monochrome display/printer
                                     adapter                  */
#define _CGA             2        /* color/graphics monitor
                                    adapter                   */
#define _HERCULES        3        /* Hercules monochrome
                                     adapter card             */
#define _MCGA            4        /* PS/2 Model 30 monitor    */
#define _EGA             5        /* enhanced graphics
                                     adapter                  */
#define _VGA             6        /* vector graphics array    */
 #define _SVGA            7        /* super VGA               */
#define _HGC            _HERCULES

#define _MONO            1        /* regular monochrome */
#define _COLOR           2        /* regular color      */
#define _ENHANCED        3        /* enhanced color     */
#define _ANALOGMONO      5        /* analog monochrome  */
#define _ANALOGCOLOR     6        /* analog color       */
```

```
#define _GROK                      0      /* no error            */
#define _GRERROR                   (-1)   /* graphics error      */
#define _GRMODENOTSUPPORTED        (-2)   /* video mode not
                                             supported           */
#define _GRNOTINPROPERMODE         (-3)   /* function n/a in
                                             this mode           */
#define _GRINVALIDPARAMETER        (-4)   /* invalid
                                             parameter(s)        */
#define _GRINSUFFICIENTMEMORY      (-5)   /* out of memory       */
#define _GRNOOUTPUT                1      /* nothing was done    */
#define _GRCLIPPED                 2      /* output clipped      */

/* Color Setting and Query Functions */

short _FAR               _setcolor( short );
short _FAR               _getcolor( void );
long _FAR                _setbkcolor( long );
long _FAR                _getbkcolor( void );
long _FAR                _remappalette( short, long );
short _FAR               _remapallpalette( long _FAR * );
short _FAR               _selectpalette( short );

#define _BLACK           0x000000L
#define _BLUE            0x2a0000L
#define _GREEN           0x002a00L
#define _CYAN            0x2a2a00L
#define _RED             0x00002aL
#define _MAGENTA         0x2a002aL
#define _BROWN           0x00152aL
#define _WHITE           0x2a2a2aL
#define _GRAY            0x151515L
#define _LIGHTBLUE       0x3F1515L
#define _LIGHTGREEN      0x153f15L
#define _LIGHTCYAN       0x3f3f15L
#define _LIGHTRED        0x15153fL
#define _LIGHTMAGENTA    0x3f153fL
#define _YELLOW          0x153f3fL
#define _BRIGHTWHITE     0x3f3f3fL
#define _LIGHTYELLOW     _YELLOW

/* Shape and Curve Drawing Functions */

short _FAR      _lineto( short, short );
short _FAR      _lineto_w( double, double );
short _FAR      _rectangle( short, short, short, short,
                            short );
```

```
short _FAR      _rectangle_w( short, double, double, double,
                              double );
short _FAR      _rectangle_wxy( short, struct _wxycoord _FAR *,
                                struct _wxycoord _FAR * );
short _FAR      _arc( short, short, short, short, short, short,
                      short, short );
short _FAR      _arc_w( double, double, double, double, double,
                        double, double, double );
short _FAR      _arc_wxy( struct _wxycoord _FAR *,
                          struct _wxycoord _FAR *,
                          struct _wxycoord _FAR *,
                          struct _wxycoord _FAR * );
short _FAR      _ellipse( short, short, short, short, short );
short _FAR      _ellipse_w( short, double, double, double,
                            double );
short _FAR      _ellipse_wxy( short, struct _wxycoord _FAR *,
                              struct _wxycoord _FAR * );
short _FAR      _pie( short, short, short, short, short, short,
                      short, short, short );
short _FAR      _pie_w( short, double, double, double, double,
                        double, double, double, double );
short _FAR      _pie_wxy( short, struct _wxycoord _FAR *,
                          struct _wxycoord _FAR *,
                          struct _wxycoord _FAR *,
                          struct _wxycoord _FAR * );
short _FAR      _polygon( short, short, struct xycoord
                          _FAR * );
short _FAR      _polygon_w( short, short, double _FAR * );
short _FAR      _polygon_wxy( short, short, struct _wxycoord
                              _FAR * ); short _FAR _floodfill
                              (short, short, short );
short _FAR      _floodfill_w( double, double, short );
short _FAR      _setpixel( short, short );
short _FAR      _setpixel_w( double, double );
short _FAR      _getpixel( short, short );
short _FAR      _getpixel_w( double, double );
short _FAR      _getarcinfo( struct xycoord _FAR *,
                             struct xycoord _FAR *,
                             struct xycoord _FAR * );

/* Position Determination Functions */

struct xycoord _FAR     _getcurrentposition( void );
struct _wxycoord _FAR   _getcurrentposition_w( void );
struct xycoord _FAR     _getviewcoord( short, short );
```

```
struct xycoord _FAR      _getviewcoord_w( double, double );
struct xycoord _FAR      _getviewcoord_wxy( struct _wxycoord
                                   _FAR * );
struct xycoord _FAR      _getphyscoord( short, short );
struct _wxycoord _FAR    _getwindowcoord( short, short );
struct xycoord _FAR      _moveto( short, short );
struct _wxycoord _FAR    _moveto_w( double, double );
struct xycoord _FAR      _setvieworg( short, short );

#define _getlogcoord     _getviewcoord       /* for
                                    compatibility */
#define _setlogorg       _setvieworg

/* Output Determination Functions */

void _FAR                _setfillmask( unsigned char _FAR * );
unsigned char _FAR * _FAR
                         _getfillmask( unsigned char _FAR * );
void _FAR                _setlinestyle( unsigned short );
unsigned short _FAR      _getlinestyle( void );
short _FAR               _setplotaction( short );
short _FAR               _getplotaction( void );

#define _setwritemode    _setplotaction      /* for
                                    compatibility */
#define _getwritemode    _getplotaction

enum {                          /* plotting action */
        _GOR, _GAND, _GPRESET, _GPSET, _GXOR
};

/* Screen Manipulation Functions */

void _FAR        _clearscreen( short );
void _FAR        _setviewport( short, short, short, short );
void _FAR        _setcliprgn( short, short, short, short );
void _FAR        _getcliprgn( short _FAR *, short _FAR *,
                              short _FAR *, short _FAR * );
short _FAR       _displaycursor( short );
short _FAR       _wrapon( short );
short _FAR       _setwindow( short, double, double, double,
                             double );

#define _GCLEARSCREEN    0
#define _GVIEWPORT       1
#define _GWINDOW         2
```

```
#define _GBORDER        2
#define _GFILLINTERIOR  3

enum {                          /* cursor display */
        _GCURSOROFF, _GCURSORON
};

enum {                          /* text wrapping */
        _GWRAPOFF, _GWRAPON
};

/* Graphics Text Manipulation Functions and Constants */

struct textsettings _FAR * _FAR
                        _gettextsettings( struct textsettings
                                          _FAR * );
void _FAR               _gettextextent( short, short, char
                                        _FAR *,
                                struct xycoord _FAR *, struct
                                xycoord _FAR * );
void _FAR               _setcharsize( short, short );
void _FAR               _setcharsize_w( double, double );
void _FAR               _settextalign( short, short );
void _FAR               _settextpath( short );
void _FAR               _settextorient( short, short );
void _FAR               _setcharspacing( short );
void _FAR               _setcharspacing_w( double );
short _FAR              _grtext( short, short, char _FAR * );
short _FAR              _grtext_w( double, double, char
                                   _FAR * );

enum {                          /* horizontal alignment */
        _NORMAL, _LEFT, _CENTER, _RIGHT
};

enum {                          /* vertical alignment */
        _TOP=1, _CAP, _HALF, _BASE, _BOTTOM
};

enum {                          /* text path */
        _PATH_RIGHT, _PATH_LEFT, _PATH_UP, _PATH_DOWN
};

/* Text Manipulation Functions */
```

```
#define _GSCROLLUP        1
#define _GSCROLLDOWN      (-1)
#define _MAXTEXTROWS      (-1)

void _FAR               _settextwindow( short, short, short,
                                        short );
void _FAR               _outtext( char _FAR * );
short _FAR              _settextcolor( short );
short _FAR              _gettextcolor( void );
struct rccoord _FAR _settextposition( short, short );
struct rccoord _FAR _gettextposition( void );
void _FAR               _scrolltextwindow( short );
void _FAR               _gettextwindow( short _FAR *, short _FAR *,
                                        short _FAR *, short
                                        _FAR * );
short _FAR              _gettextcursor( void );
short _FAR              _settextcursor( short );
void _FAR               _outmem( unsigned char _FAR *, short );
short _FAR              _settextrows( short );

/* Image Manipulation Functions */

void _FAR      _getimage( short, short, short, short, char
                          _HUGE * );
void _FAR      _getimage_w( double, double, double, double,
                            char _HUGE * );
void _FAR      _getimage_wxy( struct _wxycoord _FAR *,
                              struct _wxycoord _FAR *, char
                              _HUGE * );
void _FAR      _putimage( short, short, char _HUGE *, short );
void _FAR      _putimage_w( double, double, char _HUGE *,
                            short );
long _FAR      _imagesize( short, short, short, short );
long _FAR      _imagesize_w( double, double, double, double );
long _FAR      _imagesize_wxy( struct _wxycoord _FAR *,
                               struct _wxycoord _FAR * );

/* Pragmas required for functions having a 'double' as the
   first or second parameter */

#pragma aux graphics parm [];
#pragma aux (graphics)  _arc_w;
#pragma aux (graphics)  _ellipse_w;
#pragma aux (graphics)  _floodfill_w;
#pragma aux (graphics)  _getimage_w;
#pragma aux (graphics)  _getpixel_w;
```

```
#pragma aux (graphics)  _getviewcoord_w;
#pragma aux (graphics)  _grtext_w;
#pragma aux (graphics)  _imagesize_w;
#pragma aux (graphics)  _lineto_w;
#pragma aux (graphics)  _moveto_w;
#pragma aux (graphics)  _pie_w;
#pragma aux (graphics)  _putimage_w;
#pragma aux (graphics)  _rectangle_w;
#pragma aux (graphics)  _setcharsize_w;
#pragma aux (graphics)  _setcharspacing_w;
#pragma aux (graphics)  _setpixel_w;
#pragma aux (graphics)  _setwindow;

#undef _FAR
#undef _HUGE

#pragma pack();
#define _GRAPH_H_INCLUDED
#endif
```

Note: You may notice several places in this code where a preprocessor directive called **#pragma** is used. This directive is very compiler specific. Each compiler has a different (although similar) way of using **#pragma** directives. See the documentation for your professional compiler for further information regarding **#pragmas**.

A P P E N D I X

ANSI AND ASCII CODE CHARTS

ASCII CHARACTER CODES

This table shows the ASCII codes for the decimal, octal, hexadecimal, and character representations of the numbers from 0 through 255 (these are all the possible values for a **char** variable). ASCII is the American Standard Code for Information Interchange. It has been adopted by many computer manufacturers as a standard way of representing and recognizing characters on computers.

Remember: Character values greater than 127 have graphic codes specific to the IBM PC; these codes will not produce the same output on pure ASCII terminals. This is also true of several of the control codes (values less than 32).

Decimal	Octal	Hex	Character	Decimal	Octal	Hex	Character
0	\000	0x0		32	\040	0x20	
1	\001	0x1	☺	33	\041	0x21	!
2	\002	0x2	☻	34	\042	0x22	"
3	\003	0x3	♥	35	\043	0x23	#
4	\004	0x4	♦	36	\044	0x24	$
5	\005	0x5	♣	37	\045	0x25	%
6	\006	0x6	♠	38	\046	0x26	&
7	\007	0x7	•	39	\047	0x27	'
8	\010	0x8	◘	40	\050	0x28	(
9	\011	0x9	○	41	\051	0x29	)
10	\012	0xA	◙	42	\052	0x2A	*
11	\013	0xB	♂	43	\053	0x2B	+
12	\014	0xC	♀	44	\054	0x2C	,
13	\015	0xD	♪	45	\055	0x2D	-
14	\016	0xE	♫	46	\056	0x2E	.
15	\017	0xF	☼	47	\057	0x2F	/
16	\020	0x10	►	48	\060	0x30	0
17	\021	0x11	◄	49	\061	0x31	1
18	\022	0x12	↕	50	\062	0x32	2
19	\023	0x13	‼	51	\063	0x33	3
20	\024	0x14	¶	52	\064	0x34	4
21	\025	0x15	§	53	\065	0x35	5
22	\026	0x16	▬	54	\066	0x36	6
23	\027	0x17	↨	55	\067	0x37	7
24	\030	0x18	↑	56	\070	0x38	8
25	\031	0x19	↓	57	\071	0x39	9
26	\032	0x1A	→	58	\072	0x3A	:
27	\033	0x1B	←	59	\073	0x3B	;
28	\034	0x1C	∟	60	\074	0x3C	<
29	\035	0x1D	↔	61	\075	0x3D	=
30	\036	0x1E	▲	62	\076	0x3E	>
31	\037	0x1F	▼	63	\077	0x3F	?

Decimal	Octal	Hex	Character	Decimal	Octal	Hex	Character
64	\100	0x40	@	96	\140	0x60	`
65	\101	0x41	A	97	\141	0x61	a
66	\102	0x42	B	98	\142	0x62	b
67	\103	0x43	C	99	\143	0x63	c
68	\104	0x44	D	100	\144	0x64	d
69	\105	0x45	E	101	\145	0x65	e
70	\106	0x46	F	102	\146	0x66	f
71	\107	0x47	G	103	\147	0x67	g
72	\110	0x48	H	104	\150	0x68	h
73	\111	0x49	I	105	\151	0x69	i
74	\112	0x4A	J	106	\152	0x6A	j
75	\113	0x4B	K	107	\153	0x6B	k
76	\114	0x4C	L	108	\154	0x6C	l
77	\115	0x4D	M	109	\155	0x6D	m
78	\116	0x4E	N	110	\156	0x6E	n
79	\117	0x4F	O	111	\157	0x6F	o
80	\120	0x50	P	112	\160	0x70	p
81	\121	0x51	Q	113	\161	0x71	q
82	\122	0x52	R	114	\162	0x72	r
83	\123	0x53	S	115	\163	0x73	s
84	\124	0x54	T	116	\164	0x74	t
85	\125	0x55	U	117	\165	0x75	u
86	\126	0x56	V	118	\166	0x76	v
87	\127	0x57	W	119	\167	0x77	w
88	\130	0x58	X	120	\170	0x78	x
89	\131	0x59	Y	121	\171	0x79	y
90	\132	0x5A	Z	122	\172	0x7A	z
91	\133	0x5B	[	123	\173	0x7B	{
92	\134	0x5C	\	124	\174	0x7C	|
93	\135	0x5D	]	125	\175	0x7D	}
94	\136	0x5E	^	126	\176	0x7E	~
95	\137	0x5F	_	127	\177	0x7F	⌂

Decimal	Octal	Hex	Character	Decimal	Octal	Hex	Character
128	\200	0x80	Ç	160	\240	0xA0	á
129	\201	0x81	ü	161	\241	0xA1	í
130	\202	0x82	é	162	\242	0xA2	ó
131	\203	0x83	â	163	\243	0xA3	ú
132	\204	0x84	ä	164	\244	0xA4	ñ
133	\205	0x85	à	165	\245	0xA5	Ñ
134	\206	0x86	å	166	\246	0xA6	ª
135	\207	0x87	ç	167	\247	0xA7	º
136	\210	0x88	ê	168	\250	0xA8	¿
137	\211	0x89	ë	169	\251	0xA9	⌐
138	\212	0x8A	è	170	\252	0xAA	¬
139	\213	0x8B	ï	171	\253	0xAB	½
140	\214	0x8C	î	172	\254	0xAC	¼
141	\215	0x8D	ì	173	\255	0xAD	¡
142	\216	0x8E	Ä	174	\256	0xAE	«
143	\217	0x8F	Å	175	\257	0xAF	»
144	\220	0x90	É	176	\260	0xB0	░
145	\221	0x91	æ	177	\261	0xB1	▒
146	\222	0x92	Æ	178	\262	0xB2	▓
147	\223	0x93	ô	179	\263	0xB3	│
148	\224	0x94	ö	180	\264	0xB4	┤
149	\225	0x95	ò	181	\265	0xB5	╡
150	\226	0x96	û	182	\266	0xB6	╢
151	\227	0x97	ù	183	\267	0xB7	╖
152	\230	0x98	ÿ	184	\270	0xB8	╕
153	\231	0x99	Ö	185	\271	0xB9	╣
154	\232	0x9A	Ü	186	\272	0xBA	║
155	\233	0x9B	¢	187	\273	0xBB	╗
156	\234	0x9C	£	188	\274	0xBC	╝
157	\235	0x9D	¥	189	\275	0xBD	╜
158	\236	0x9E	Pt	190	\276	0xBE	╛
159	\237	0x9F	ƒ	191	\277	0xBF	┐

Decimal	Octal	Hex	Character	Decimal	Octal	Hex	Character
192	\300	0xC0	└	224	\340	0xE0	α
193	\301	0xC1	┴	225	\341	0xE1	β
194	\302	0xC2	┬	226	\342	0xE2	Γ
195	\303	0xC3	├	227	\343	0xE3	π
196	\304	0xC4	─	228	\344	0xE4	Σ
197	\305	0xC5	┼	229	\345	0xE5	σ
198	\306	0xC6	╞	230	\346	0xE6	µ
199	\307	0xC7	╟	231	\347	0xE7	τ
200	\310	0xC8	╚	232	\350	0xE8	ϕ
201	\311	0xC9	╔	233	\351	0xE9	θ
202	\312	0xCA	╩	234	\352	0xEA	Ω
203	\313	0xCB	╦	235	\353	0xEB	δ
204	\314	0xCC	╠	236	\354	0xEC	∞
205	\315	0xCD	═	237	\355	0xED	ø
206	\316	0xCE	╬	238	\356	0xEE	∈
207	\317	0xCF	╧	239	\357	0xEF	∩
208	\320	0xD0	╨	240	\360	0xF0	≡
209	\321	0xD1	╤	241	\361	0xF1	±
210	\322	0xD2	╥	242	\362	0xF2	≥
211	\323	0xD3	╙	243	\363	0xF3	≤
212	\324	0xD4	╘	244	\364	0xF4	⌠
213	\325	0xD5	╒	245	\365	0xF5	⌡
214	\326	0xD6	╓	246	\366	0xF6	÷
215	\327	0xD7	╫	247	\367	0xF7	≈
216	\330	0xD8	╪	248	\370	0xF8	°
217	\331	0xD9	┘	249	\371	0xF9	•
218	\332	0xDA	┌	250	\372	0xFA	·
219	\333	0xDB	█	251	\373	0xFB	√
220	\334	0xDC	▄	252	\374	0xFC	η
221	\335	0xDD	▌	253	\375	0xFD	²
222	\336	0xDE	▐	254	\376	0xFE	■
223	\337	0xDF	▀	255	\377	0xFF	

ANSI SCREEN-HANDLING CODES

Table B-1 shows the ANSI codes for screen control. ANSI is the American National Standards Institute, a body of people who generate standards for the world to follow. Many computer manufacturers support the ANSI standard for screen manipulation as presented in Tables B-1 and B-2.

Remember that each ANSI code begins with the characters <ESC>[, which can be printed to the screen using the following **printf( )** statement:

```
printf("%c[", 27);
```

For examples of using the ANSI screen controls, refer to the **SayANSI( )** function in Chapter 8.

Command	Sequence	Function
Cursor Position	ESC[r;cH	Moves to the screen position noted as row *r* and column *c*. If no values are given (ESC[H), cursor will move to home position (top left of screen)
Cursor Up	ESC[nA	Moves the cursor up *n* lines. If *n* is not specified, moves up 1 row
Cursor Down	ESC[nB	Moves the cursor down *n* lines. If *n* is not specifed, moves down 1 row
Cursor Forward	ESC[nC	Moves the cursor *n* spaces to the right on the screen. If *n* is not specified, moves 1 space to the right. Will ignore movement past the right edge of the screen

TABLE B-1
ANSI Screen Control Commands

Command	Sequence	Function
Cursor Backward	ESC[nD	Moves the cursor *n* spaces to the left on the screen. If *n* is not specified, moves 1 space to the left. Will ignore movement past the left edge of the screen
Horizontal and Vertical Position	ESC[r;cf	Moves to screen position noted as row *r* and column *c*. If no values are given (ESC[H), cursor will move to the home position (top left of screen). Works the same as Cursor Position
Save Cursor Position	ESC[s	Saves the current cursor position, for later restoration.
Restore Cursor Position	ESC[u	Restores the cursor position, which must have been previously saved.
Erase Display	ESC[xJ	Clears portions of the screen, based on the value of *x*. If *x* is 0 or unspecified, screen is cleared from current cursor location to the end. If *x* is 1, screen is cleared from home position up to current cursor position. If *x* is 2, entire screen is cleared and cursor is moved to home position
Erase Line	ESC[xK	Clears portions of current line, based on the value of *x*. If *x* is 0 or unspecified, line is cleared from current cursor location to end. If *x* is 1, line is cleared from first column through current cursor position. If *x* is 2, entire line is cleared and cursor is moved to leftmost position on line
Set Rendition	ESC [n;...;nm	Sets the attributes with which text is placed on the screen. Zero or more *n* values may be used in the command, separated by semicolons. Possible values for *n* are in Table B-2. If no values are specified (ESC[m), a default of 0 (reset) is assumed

TABLE B-1
ANSI Screen Control Commands (continued)

Value	Description
0	Reset; all attributes turned off. Often placed at beginning of each set of attributes, to turn off any previous settings (see the **SayColor( )** function in Chapter 8). For instance, to set attributes to high-intensity white foreground on blue background, you might use ESC[0;1;37;44m
1	High-intensity
4	Underline
5	Blinking
7	Inverse video (swaps foreground and background colors)
8	Invisible display (black on black); useful for passwords or other confidential input
30	Black foreground
31	Red foreground
32	Green foreground
33	Yellow foreground
34	Blue foreground
35	Magenta foreground
36	Cyan foreground
37	White foreground
40	Black background
41	Red background
42	Green background
43	Yellow background
44	Blue background
45	Magenta background
46	Cyan background
47	White background

TABLE B-2
ANSI Rendition Attribute Values

BIBLIOGRAPHY

There are probably more books about the C language than about any other single computer topic, and this bibliography is not meant to be exhaustive by any means. Listed here are books that are useful in learning about C and software development, and are recommended for ongoing reference needs. The materials are divided into three broad topics: General Software Development, C Programming Guides, and Programming for the IBM PC. The final portion of this bibliography lists several good programming magazines.

GENERAL SOFTWARE DEVELOPMENT

Boddie, John. *Crunch Mode—Building Effective Systems on a Tight Budget.* Englewood Cliffs, NJ: Yourdon Press, 1987.

Campbell, Sally. *Microcomputer Software Design—How to Develop Complex Application Programs.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

DeGrace, Peter, and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions—A Catalogue of Modern Software Engineering Paradigms.* Englewood Cliffs, NJ: Yourdon Press, 1990.

DeMarco, Tom, and Timothy Lister. *Peopleware—Productive Projects and Teams.* New York: Dorset House Publishing, 1987.

Glass, Robert L. *Software Conflict—Essays on the Art and Science of Software Engineering.* Englewood Cliffs, NJ: Yourdon Press, 1991.

Lammers, Susan. *Programmers at Work.* Redmond, WA: Microsoft Press, 1986.

von Mayrhauser, Anneliese. *Software Engineering—Methods and Management.* San Diego, CA: Academic Press, 1990.

Weinberg, Gerald M. *Understanding the Professional Programmer.* New York: Dorset House Publishing, 1988.

C PROGRAMMING GUIDES

Campbell, Joe. C Programmer's Guide to Serial Communications. Indianapolis, IN: Howard W. Sams & Company, 1987.

Ferraro, Richard F. *Programmer's Guide to the EGA and VGA Cards.* Reading, MA: Addison-Wesley, 1988.

Hansen, Augie. *Proficient* C. Redmond, WA: Microsoft Press, 1987.

Hunt, William James. *The C Toolbox.* Reading, MA: Addison-Wesley, 1985.

Johnson, Nelson. *Advanced Graphics in* C. Berkeley, CA: Osborne/McGraw-Hill, 1987.

Schildt, Herbert. *The Art of C: Elegant Programming Solutions.* Berkeley, CA: Osborne/McGraw-Hill, 1991.

Schildt, Herbert. *Born To Code in* C. Berkeley, CA: Osborne/McGraw-Hill, 1989.

Schildt, Herbert. *C: The Pocket Reference.* Berkeley, CA: Osborne/McGraw-Hill, 1988.

Schustack, Steve. *Variations in* C. Redmond, WA: Microsoft Press, 1985.

Schwaderer, W. David. *C Programmer's Guide to NetBIOS.* Indianapolis, IN: Howard W. Sams & Company, 1988.

Stevens, Roger T. *Graphics Programming in* C. Redwood City, CA: M&T Books, 1989.

PROGRAMMING FOR THE IBM PC

Dettman, Terry. *DOS Programmer's Reference, 2nd Edition.* Carmel, IN: Que Corporation, 1989.

Jourdain, Robert. *Programmer's Problem Solver for the IBM PC, XT & AT.* New York: Brady Communications Company, 1986.

MAGAZINES

C Gazette: The Code-Intensive C and C++ Magazine for MS-DOS, OS/2 and Windows. P.O. Box 70167, Eugene, OR 97401-9772. A very good C programming magazine, with lots of code in each issue.

Computer Language. P.O. Box 51258, Boulder, CO 80321-1258. A good magazine for programmers of any language, and many of the articles are about C.

The C Users Journal. 1601 West 23rd Street, Suite 200. Lawrence, KS 66046-4100. A good magazine for anyone interested in C programming. Includes articles for both beginning and advanced programmers. Also has a "C Bookstore" and a disk library.

Dr. Dobb's Journal: Software Tools for the Professional Programmer. P.O. Box 52226, Boulder, CO 80321-2226. Very good programmer's magazine, with lots of code in C and other languages.

PC Techniques. 7721 E. Gray Road, Suite 204, Scottsdale, AZ 85260-9747. A wonderful magazine for programmers using the IBM PC platform. Code in C and other languages, with lots of tips for programming the PC.

Programmer's Journal. P.O. Box 70387, Eugene, OR 97401-9799. Another good programmer's magazine. Coverage of all major PC programming languages, but lots of C-based examples. Currently out of print, but back issues are worth looking for.

C [illegible]

INDEX

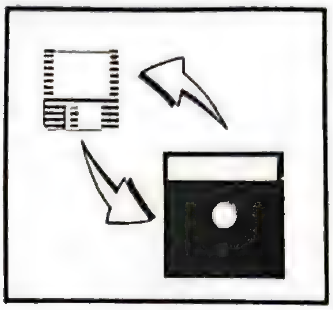
B

C

D

E

Replacement Disk Order Form



The disk supplied in C *DiskTutor* is a 3 1/2" high density (1.44 megabyte) disk. If you require a different size disk, use the order form below to order your replacement set.

Please include your original 3 1/2" disk, along with $2.00 to cover shipping and handling. Be sure to supply all of the following information:

Name: ______________________________
Company: ______________________________
Street Address: ______________________________
____
City: ______________________________
State/Province: ______________________________
ZIP/Postal Code: ______________________________
Country: ______________________________

IMPORTANT:

Size of Disk Required: ☐ 3.5" 720K ☐ 5.25" 360K ☐ 5.25" 1.2M

Mail Your Request To:

CDS Disk Fulfillment Service
ATTN: C DiskTutor Replacement
P. O. Box 7549
York, PA 17404

DiskTutor Environment — Upgrade Form

The disk supplied with this book contains DTE.EXE, a small version of M3/The Picasso Programmer's Development package, available from PSGL. The full version of this program, including manuals, disk, and a year of updates is normally $99, but as an owner of C *DiskTutor*, you may upgrade for only $39. The full version of the M3 program includes many enhanced features for developing code in any language, using any compiler. Other functions available from within the environment include source code maintenance, source code printing, unlimited open files, macros, and much more. To order, send a check or money order for $45 US ($39 plus $6 shipping) to the address below, and be sure to answer the following questions.

Name: ______________________________

Company: ______________________________

Street Address: ______________________________

____

City: ______________________________

State/Province: ______________________________

ZIP/Postal Code: ______________________________

Country: ______________________________

IMPORTANT:

Size of Disk Requested: ☐ 3.5" 720K ☐ 5.25" 360K ☐ 5.25" 1.2M ☐ 3.5" 1.44

Mail Your Request To:

Picasso Software Group Ltd.
ATTN: C DiskTutor Upgrade
P. O. Box 7549
York, PA 17404

☐ Please send me more information about M3. I'm not ready to order yet.

Upgrade to the Complete WATCOM C Package

Included with this book is a limited, introductory version of WATCOM C for running the example programs. You can obtain the full WATCOM C package at a special price. Call or fax us at the numbers below for further details.

The full WATCOM C compiler is a professional programming package that enables development and debugging of 16-bit applications for DOS, Windows™, and OS/2™. It is designed to produce high-performance optimized code that is fast, tight, and reliable.

When you upgrade to the full WATCOM™ C package, you'll get these key features:

- Support for 6 memory models: tiny, small, medium, compact, large, huge
- Interactive, source-level debugger
- Profiler for performance tuning
- Protected-mode compiler and linker (uses 386™ system for increased capacity)
- C Run-time library (Windows and OS/2 versions)
- Components licensed from the Microsoft® Windows SDK
- MAKE utility
- Object code librarian
- Complete documentation
- No-charge run-time redistribution

Call, fax or write:

415 Phillip Street
Waterloo, Ontario, Canada
N2L 3X2
tel: **1-800-265-4555** (toll-free in North America)
tel: (519) 886-3700
fax: (519) 747-4971

Send me more information about the WATCOM C Upgrade

Name ____

Company ____

Address ____

____

City ____________________ State ______________ Zip __________

Phone ______________________ Fax ______________________

C

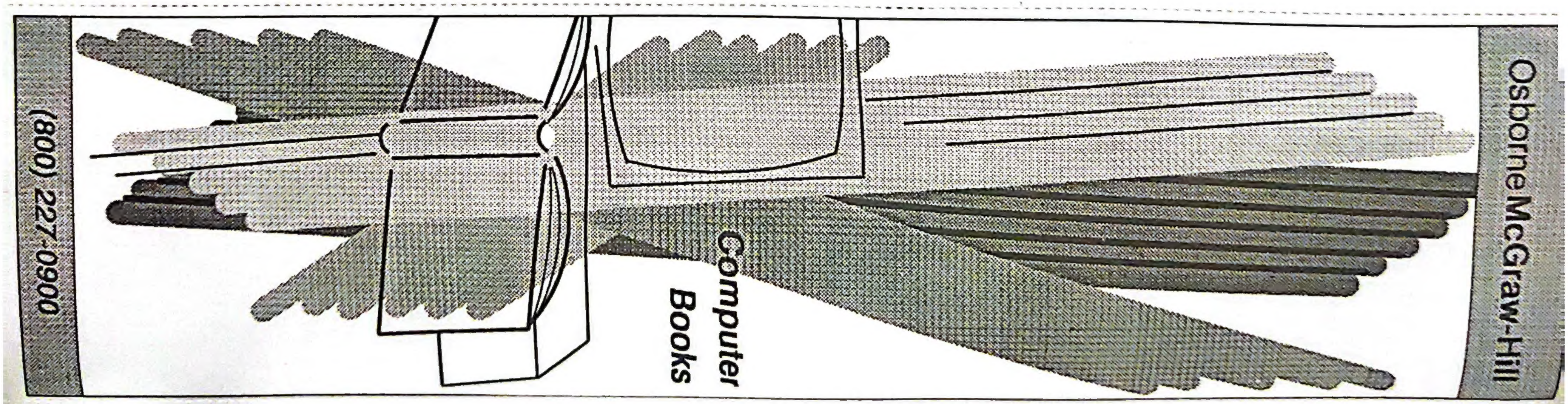
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

First Class Permit NO. 3111 Berkeley, CA

Postage will be paid by addressee

Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710–9938



About the Author

John Ribar is a programmer and the author of several acclaimed books on C programming as well as many shareware programs. He is a consultant for St. Onge Company, a material handling engineering firm in York, Pennsylvania. Ribar lives in York with his wife and four children.

Covers Standard ANSI C for All IBM PC Compatible Computers

IBM PC & Compatibles/Programming Languages

Skill Level Guide
- ☒ Beginning
- ☒ Intermediate
- ☐ Advanced
- ☐ All of the above

C DISKTUTOR

THE DISK - Includes a Special Version of the WATCOM C Compiler with a Programming Environment on One 3.5-Inch Disk

For Hardware Requirements, See Inside Front Cover

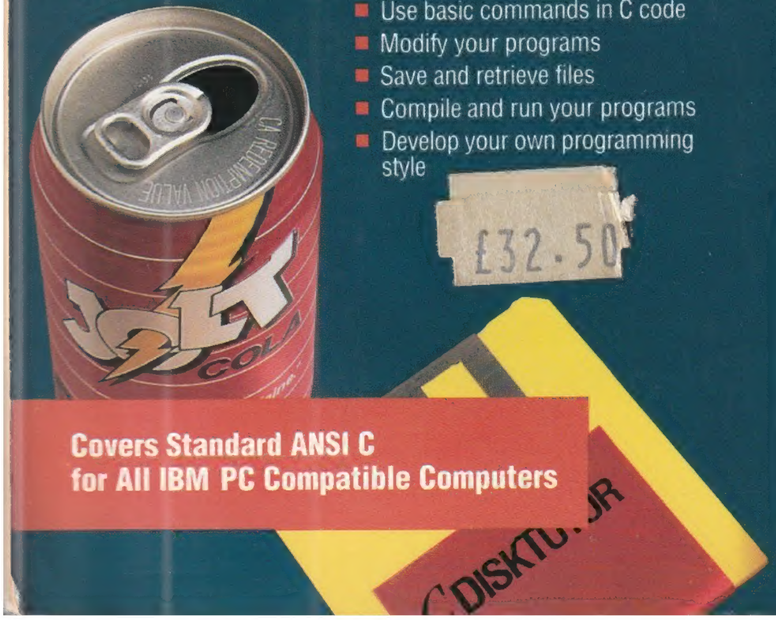
You're looking at an exceptional book/disk package that includes everything you need to learn how to program in standard ANSI C quickly and easily. Unlike other introductory C programming books, **C DISKTUTOR** offers

- An excellent self-teaching guide to C
- A library of useful functions which you'll use again and again in your own applications
- A special version of the acclaimed WATCOM C compiler with a programming environment that you can use to write effective programs
- 3 full-blown case studies and 8 mini-case studies to help you apply your growing knowledge of C

If you're new to programming, you'll learn the fundamentals of the C language, step-by-step. Just load the WATCOM C compiler and follow the examples in the book and on the disk, and you'll begin to write your first program.

Ribar thoroughly explains how to

- Install the WATCOM compiler and the C DISKTUTOR programming environment
- Use basic commands in C code
- Modify your programs
- Save and retrieve files
- Compile and run your programs
- Develop your own programming style



Covers Standard ANSI C for All IBM PC Compatible Computers

Once you've got the beginning concepts down, Ribar teaches you about

- Program design
- Data storage and manipulation
- Controlling program flow
- Writing functions to modernize your work
- The C preprocessor
- The C function library
- Designing reusable program libraries
- The similiarities and differences between Pascal and C with special "From Pascal to C" examples

If you're already programming in C but want to expand your skills, you can easily take the C programs presented in the book and customize them for your own purposes before adding them to your own repertoire.

Throughout the book, Ribar presents case studies that offer real-world C programming solutions. You'll find useful programming projects that you can incorporate into your daily routine such as

- A programmer's calculator
- A file dump utility
- An electronic address book
- A text-based user interface library

Learn at your own pace and soon you'll be writing full-fledged C programs that not only get the job done, but are also simple to customize and maintain long after you've finished reading this book.

All in all, C DISKTUTOR is by far the best hands-on introduction you can find anywhere to the C programming language.

ISBN 0-07-881798-6
53995 >
9 780078 817984

$39.95